MICRO APPLICATION

PROGRAMMER EN LANGAGE C

- Les compilateurs Turbo C et Quick C parfaitement détaillés.
- De nombreux exercices pour faire le point sur vos connaissances.

AVEC LA DISQUETTE 3" 1/2

UN LIVRE DATA BECKER

ma

Albrecht Darimont

Programmer en Langage C

EDITIONS MICRO APPLICATION

Copyright © 1992 DATA Becker GmbH
Merowingerstraße, 30
4000 Düsseldorf

© 1993 Micro Application
58, rue du Faubourg Poissonnière
75010 Paris

Auteur Albrecht Darimont

Traducteur Dominique Kocher

'Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de MICRO APPLICATION est illicite (Loi du 11 Mars 1957, article 40, 1er alinéa).

Cette représentation ou reproduction illicite, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal.

La Loi du 11 Mars 1957 n'autorise, aux termes des alinéas 2 et 3 de l'article 41, que les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à l'utilisation collective d'une part, et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration'.

ISBN : 2-86899-845-3

Collection dirigée par Philippe Olivier
Edition réalisée par Frédérique Beaudonnet

MS-DOS, Microsoft et Microsoft Quick C sont des marques déposées de Microsoft Corporation.
Turbo C++ et Turbo C sont des marques enregistrées de Borland International Inc.
Turbo C++ et Turbo C sont des produits de la société Borland qui en détient le copyright à l'échelon mondial.
Ce livre n'est pas le manuel des logiciels Turbo C++ et Turbo C et son contenu n'engage pas la société Borland.
Tous les autres produits sont des marques déposées de leurs sociétés respectives.

Sommaire

1 Introduction 11

1.1. Installation des compilateurs 14
1.2. L'environnement de développement 17
1.3. Initiation rapide : éditer, compiler, lancer 20

2 Premiers pas en C 23

2.1. Le plus petit programme C possible 24
2.2. Un programme avec sortie sur l'écran 25
2.3. Travailler avec le compilateur 26
2.3.1. Edition de programmes C 27
2.3.2. Compilation de programmes C 27
2.3.3. Lancement de programmes 29
2.3.4. Correction d'erreurs 29
2.4. Programmation de renvois de texte 30
2.5. Renvoi de nombres 33
2.6. Intégration de commentaires dans des programmes C 35
2.7. Récapitulation 36

3 Structures de contrôle en C 39

3.1. Boucles testées en entrée à l'aide de while 40
3.2. Boucles testées en sortie à l'aide de do..while 42
3.3. Boucles testées en entrée à l'aide de for 43
3.4. Structure de renvoi - L'instruction if 45
3.5. Interruption de boucle - Break et Continue 47
3.6. Différenciations de cas à l'aide de Switch 49
3.7. Renvoi inconditionnel à l'aide de Goto 52
3.8. Structures de contrôle liées 55
3.9. Récapitulation 58

4 Affectations de valeurs et comparaisons logiques 61

4.1. Affectation de valeurs entières 61
4.2. Autres types de données 63
4.3. Conversion de variables 65
4.4. Types de données supplémentaires 66
4.5. Comparaisons logiques 68
4.5.1. Valeurs de vérité 71
4.5.2. Liaisons logiques 72
4.5.3. Evaluation de comparaisons logiques 73
4.5.4. Priorités d'évaluation 73
4.6. Récapitulation 74

5 Les opérateurs spéciaux en C 77

5.1. Opérateur arithmétique 79
5.2. Structure if compacte 80
5.3. Récapitulation 81

6 Fonctions 83

6.1. Une première fonction - Calcul de sommes et de carrés . . . 85
6.2. Arguments de fonctions 87
6.3. Retourner des résultats de fonctions à l'aide de Return 88
6.4. Fonctions à virgule flottante 90
6.5. Domaines de définition de variables 93
6.5.1. Variables automatiques ou locales 96
6.5.2. Variables de type registre 96
6.6. Prototypes 97
6.7. Bibliothèques de fonctions standard 98
6.8. Récursivité 98
6.9. Gestion de la mémoire sur un exemple de récursion 101
6.9.1. Débogage à l'aide du débogueur de Turbo C 2.0 et C++ 3.0 104
6.9.2. Débogage à l'aide du débogueur de Quick C 2.5 108
6.10. Récapitulation 112

7 Constantes et macros 115

7.1. Définir des macros 117
7.2. Ce qu'il ne faudrait pas faire 118
7.3. Récapitulation 119

8 Chaînes et tableaux 121

8.1. Traitement d'une chaîne 122
8.1.1. Renvoi de chaînes partielles 123
8.1.2. Copier des chaînes 124
8.1.3. Comparer des chaînes 126
8.1.4. Fusionner des chaînes 126
8.2. Qu'est-ce qu'un tableau ? 127
8.2.1. Tableau de valeurs entières 127
8.2.2. Initialiser des tableaux 128
8.2.3. Tableaux pluridimensionnels 129
8.3. Contrôle logique à l'aide du débogueur (II) 134
8.4. Récapitulation 135

9 Transmission de données à des fonctions 137

9.1. Modifier des données externes 138
9.2. Récapitulation 140

10 Pointeurs 141

10.1. Deux règles importantes 142
10.2. Deux pointeurs pour une variable 143
10.3. Déclarer des pointeurs 145
10.4. Pointeurs et tableaux (arrays) 146
10.5. Pointeur dirigé sur des valeurs entières 149
10.6. Transmission de données à des fonctions à l'aide de pointeurs 149
10.7. Initialiser des tableaux de pointeurs (pointerarrays) 151
10.8. Récapitulation 154
10.9. Un exemple pour bien comprendre 155

11 Entrées/Sorties standard 159

11.1. Lire des caractères 161
11.1.1. Saisie par l'intermédiaire du buffer de MS-DOS 162
11.1.2. Saisie de données sans passer par le buffer 163
11.1.3. Saisie avec saut de ligne programmé 164
11.2. Lecture de valeurs entières 166
11.3. Erreurs avec scanf() 168
11.4. Lecture de chaînes 169
11.5. Lecture de chaînes de longueur quelconque 170
11.6. Entrées/Sorties dans la mémoire de travail 172
11.7. Récapitulation 174

12 Gestion des fichiers en C 177

12.1. Ecrire dans un fichier 178
12.1.1. Ecrire des chaînes 178
12.1.2. Ajouter des caractères 180
12.2. Lecture d'un fichier 181
12.2.1. Lecture de caractères 182
12.2.2. Lecture de mots 183
12.2.3. Lecture de lignes 185
12.3. Ouverture et lecture de fichiers quelconques 186
12.4. Transmission de noms de fichiers à main() 187
12.5. Impression du contenu d'un fichier 190
12.6. Récapitulation 191

13 Types de données complexes 195

13.1. Structures 195
13.1.1. Affectations de valeurs à des structures 198
13.1.2. Affichage de champs de structures 198
13.2. Un tableau de structures 199
13.3. Pointeurs et structures 200
13.4. Arithmétique des pointeurs 202
13.5. Structures nommées et imbriquées 203
13.6. Affectations de valeurs et attribution de noms 206
13.7. Le type de données union 207
13.8. Définir des types de données à l'aide de typedef 210

13.9. Ecrire et lire des structures 211
13.9.1. Ecriture d'enregistrements dans un fichier 212
13.9.2. Extraction d'enregistrements d'un fichier à l'aide de fread() 215
13.9.3. Recherche et affichage sélectif de données 217
13.9.4. Modifier des enregistrements 223
13.10. Récapitulation 228

14 Gestion dynamique de la mémoire et traitement de listes 231

14.1. Créer des variables dynamiques sur le tas 234
14.2. Allocation dynamique de la mémoire à l'aide de malloc() . . 235
14.3. Listes à simple chaînage 237
14.3.1. Construction 237
14.3.2. Le premier élément d'une liste 238
14.3.3. Les autres éléments d'une liste 238
14.3.4. Accès aux éléments d'une liste 240
14.3.5. Libération des variables dynamiques 241
14.4. Listes à double chaînage 241
14.4.1. La fonction append()
- Ajouter des éléments en fin de liste 247
14.4.2. La fonction insert()
- Insérer des éléments dans une liste 248
14.4.3. La fonction delete()
- Supprimer des éléments d'une liste 251
14.4.4. Les fonctions croissant() et decroissant()
- Défilement d'une liste 252
14.5. Récapitulation 253

15 Manipulation de caractères et de bits 255

15.1. Classification de caractères 255
15.2. Opérateurs logiques pour la manipulation de bits 259
15.3. L'opérateur Shift 262
15.4. Accès individuel à des bits 264
15.5. Récapitulation 266

16 C, Pascal, BASIC - En quoi diffèrent-ils ? 269

16.1. Différences fondamentales avec le BASIC 270

16.2. Différences fondamentales avec le Pascal 274

16.2.1. Types de données et opérateurs 274

16.2.2. Opérateurs d'affectations 278

16.2.3. Structures de contrôle 279

16.3. Erreurs de programmation fréquentes 286

17 Annexes 289

17.1. Les mots clés du C 289

17.2. Les opérateurs en C 290

17.3. Types de données en C 291

17.4. Structures de contrôle en C 291

17.5. Récapitulation des syntaxes 293

17.6. Commandes d'édition sous Turbo C et C++ 304

17.7. Commandes d'édition sous Quick C 305

17.8. Affectation des touches de fonction en Turbo C et C++ . . . 306

17.9. Affectation des touches de fonction en Quick C 306

17.10. Solutions des exercices 307

Index 331

1 Introduction

Vous avez dit "C"?

Petite devinette. Quel est le langage de programmation sur PC dont la seule évocation fait miroiter de plaisir l'oeil du développeur ? Quel est le seul outil de développement capable d'arborer fièrement sur son revers de veston les insignes "vitesse", "compacité" et "portabilité" ? Quel est le seul langage synonyme de "conception de logiciels professionnels sur PC" ?... Ne soufflez plus ! C'est le C.

L'histoire du C commence vers la fin des années 60, dans les laboratoires Bell du New Jersey, un centre de recherche parmi les plus en pointe dans le domaine des communications. C'est ici que Ken Thompson développa le langage de programmation B, que Dennis R. fera évoluer en C vers 1972. Initialement, C fut conçu pour les ordinateurs de milieu de gamme. Son cahier des charges se résumait en deux mots : la puissance et le meilleur compromis possible sur le terrain de la portabilité (facilité de porter une application d'un système informatique à l'autre). Un langage à la fois très proche du processeur, avec le bénéfice de la puissance, et néanmoins suffisamment distant pour épargner aux développeurs les corvées de la programmation linéaire (en opposition avec la programmation structurée comme celle de Turbo Pascal).

Avec de tels arguments de séduction, C est devenu un langage très prisé dans la communauté des programmeurs, toutes machines confondues. Mais il faut bien constater que c'est dans le macrocosme des micro-ordinateurs PC qu'il a littéralement conquis les masses.

Les programmes rédigés en C sont extrêmement rapides. De fait, la majorité des éditeurs de logiciels professionnels conçoivent exclusivement leurs applications sous ce langage. Autre atout du C, son haut niveau de standardisation. Reconnu par l'American National Standard Institute (ANSI), C a fait l'objet d'une normalisation massive, portant aussi bien sur les techniques de programmation, que sur l'homogénéité de la syntaxe. Voilà pourquoi le transfert de programmes C sur d'autres types d'ordinateurs avec d'autres systèmes d'exploitation nécessite très peu d'intervention de la part des programmeurs.

A propos de cet ouvrage...

Si vous avez sous la main un PC sous MS-DOS avec une mémoire vive de 512 kilo-octets et un disque dur (recommandé) ; si vous possédez ou comptez acquérir incessamment un langage comme Turbo C, Turbo C++ (de Borland), Quick C (de Microsoft), ou d'autres outils similaires, alors voici le livre qu'il vous faut.

Il vous présentera tous les aspects de la programmation en langage C. A partir de cours théoriques, bien sûr, mais en donnant aux exemples de programmes et aux exercices corrigés une importance capitale. Dans le cas d'un langage sophistiqué comme le C, rien ne vaut la pratique. C'est pourquoi nous vous proposerons à la fin de chaque chapitre des exercices corrigés dont les solutions sont listées en fin d'ouvrage. Un bon moyen de revoir et d'approfondir vos acquis.

Le livre s'articule autour de quatre axes, dont une partie annexe qui sera votre guide de référence.

La première partie, qui regroupe les chapitres 1 à 4, traite de tous les concepts fondamentaux de la programmation en C.

Dans la deuxième partie, qui s'étend des chapitres 5 à 10, le champ de vos connaissances s'élargira sensiblement. Vous y découvrirez le principe de déclaration des champs de données, et surtout les techniques de travail propres aux pointeurs, notions indispensables à la programmation en C.

La troisième partie (les chapitres 11 et 12) est consacrée à la programmation des entrées/sorties ainsi qu'au traitement des données.

Dans la quatrième partie, vous attaquerez la programmation des structures de données complexes, la gestion dynamique de la mémoire ou le traitement des listes. Le chapitre 15 vous initiera à la manipulation des bits, et à ses multiples

possibilités, tandis que le chapitre 16 s'attachera en guise d'épilogue à comparer ces trois standards de la programmation que sont le C, le Pascal et le BASIC.

En annexe, vous trouverez un guide de langage complet, regroupant les mots-clés du C, les opérateurs, les types de données et les structures de contrôle. Par ailleurs est décrite la syntaxe de toutes les fonctions écrites dans ce livre. Enfin vous profiterez d'une vue d'ensemble des commandes d'édition propres aux compilateurs C cités dans ce livre.

Conventions typographiques

Un certain nombre d'icônes sont intégrées au texte. Elles sont destinées à attirer votre attention sur des conseils, des techniques, et aussi des définitions.

Les règles et définitions signalées par cette icône ne concernent que le langage Microsoft Quick C.

Les règles et définitions signalées par cette icône ne concernent que les langages Borland Turbo C et C++.

Les règles et définitions signalées par cette icône ne concernent que Turbo C et Quick C.

Le texte qui suit cette icône est valable pour tous les compilateurs C.

Attire votre attention sur des astuces et techniques qui simplifient la programmation.

Remarques concernant des sources d'erreurs possibles, explications se rapportant à des sections de programmes, etc...

Définitions portant sur la syntaxe des diverses commandes, explications touchant à des structures de programmes, etc...

Les explications signalées par cette icône se rapportent à la mise en pratique des exemples.

Do it! Vous invite à saisir les listings présentés, et à mettre vos connaissances théoriques en pratique.

Dans le texte, toutes les désignations de fonctions et mots-clés du C sont systématiquement écrits en italique. Les noms de fichiers et les commandes MS-DOS sont imprimés en majuscule. Les lignes de programmation, les options de menus et les textes renvoyés à l'écran sont imprimés dans le format suivant :

```
printf("Bonjour tout le monde");
```

Dans le texte, les programmes de démonstration sont imprimés avec des numéros de lignes. Ces numéros sont destinés à faciliter votre orientation dans le texte. Ils ne font pas partie du texte du programme, et il convient donc de ne pas les saisir. Lorsque vous entrerez le texte d'un programme, toutes les lignes associées à un même numéro devront être écrites dans une seule et même ligne. Un nom de fichier est rappelé dans tous les listings. Celui-ci servira de référence lorsque, par la suite, vous serez invité à vous reporter à ces listings. Par conséquent, lorsque vous entrez un tel listing et que vous le sauvegardez, nous vous conseillons vivement de l'enregistrer sous le nom proposé dans le texte même du programme.

Au sein de ce chapitre, vous apprendrez à installer les compilateurs dont il sera question. Y seront abordées les fonctions de base des environnements de développement de Microsoft Quick C 2.5, Borland Turbo C 2.0, et Turbo C++ 3.0. Ces présentations seront agrémentées d'un programme de démonstration simple.

1.1. Installation des compilateurs

Les compilateurs Turbo C 2.0 et C++ s'installent par l'intermédiaire du programme d'installation INSTALL.EXE qui se trouve sur la disquette 1 originale. Entendez par "installation" que tous les fichiers nécessaires au fonctionnement du compilateur seront copiés sur le disque dur. Dans les paragraphes qui suivent, nous décrivons l'installation de Turbo C 2.0. Les diverses étapes de cette procédure sont également applicables à l'installation de C++.

Introduisez la disquette 1 originale dans le lecteur A. Activez cette unité par

```
A:<Return>
```

et entrez la commande

```
INSTALL <Return>
```

Le programme d'installation vous demande d'entrer la désignation de l'unité à partir de laquelle vous lancez le programme d'installation (soit A dans notre cas).

Dans l'écran qui suit, validez la première option "Installation de Turbo C sur disque dur". Vous allez à présent indiquer le répertoire où sera installé Turbo C, les fichiers librairies (bibliothèques) et objets, les fichiers Header, les fichiers nécessaires à l'installation de BGI (Borland Graphics Interface), les fichiers exemples. Vous pourrez également choisir les modèles mémoire à installer sur votre disque. Amenez la barre de sélection sur la première option et appuyez sur <Return>. Entrez un autre nom de répertoire par exemple C:\TURBOC ou validez celui proposé, C:\TC. Si vous indiquez un autre répertoire, toutes les autres lignes seront automatiquement modifiées.

Pour lancez l'installation, il vous suffira de sélectionner l'option correspondante. Dès cet instant, les autres étapes de l'installation défileront automatiquement.

Lorsque le compilateur est installé, vous pourrez encore modifier les chemins de recherche pour les bibliothèques et les fichiers Include. Passez à cet effet dans le répertoire du compilateur à l'aide de

```
CD \TC <Return>
```

puis lancez le compilateur par saisie de

```
TC <Return>
```

Appuyez à présent sur la combinaison de touches <Alt>+<O>. Actionnez ensuite la touche <D>. Suite à cela, vous vous trouvez dans le menu *Options/Directories*. Appuyez sur <I>, et entrez le chemin de recherche dans le champ de saisie prévu à cet effet.

Passez alors dans le champ de saisie prévu pour le chemin de recherche des fichiers de bibliothèques en appuyant sur <L>, et entrez un nouveau chemin.

Une pression sur la touche <Esc> vous ramène dans le menu *Options*. Appuyez sur <S> (=*Save options*), et confirmez le nom de fichier proposé par le compilateur, soit TCCONFIG.TC, par pression sur <Return>.

L'installation est achevée, et vous pouvez commencer à travailler avec le compilateur.

L'installation de Quick C 2.5 s'effectue, elle aussi, par l'intermédiaire d'un programme d'installation. Introduisez la disquette 1 dans l'unité A:. Activez ce lecteur par

```
A: <Return>
```

puis entrez cette fois la commande

```
SETUP <Return>
```

Ce programme vous demande de spécifier l'unité dans laquelle se trouvent les disquettes d'installation (soit A:, dans notre cas) ainsi que la désignation du disque dur sur lequel vous désirez installer le compilateur, et il vous invite à indiquer si un coprocesseur mathématique est installé sur votre machine, ainsi que les modèles mémoire que vous désirez installer. Vous pouvez valider les propositions par défaut que le programme d'installation affiche entre crochets en appuyant sur <Return>, ou choisir vos propres paramètres. Dans ce dernier cas, <N> signifie "ne pas installer", et <O> signifie "installer".

Pour travailler avec ce livre, nous vous suggérons d'accepter les valeurs proposées par défaut dans les programmes d'installation, et d'installer de plus les modèles mémoire MEDIUM, COMPACT et LARGE/HUGE par saisie de <O>.

Dans une deuxième phase, il vous est demandé de préciser les répertoires dans lesquels le compilateur doit être copié. Contentez-vous de valider les noms de répertoires proposés en appuyant sur <Return>.

Dès que vous avez répondu à toutes les questions concernant l'installation, celle-ci se déroule automatiquement. Il vous suffit d'introduire dans le lecteur les disquettes demandées par le programme. Mais avant de pouvoir lancer le compilateur, il vous faut activer le programme NEW-VARS.BAT qui permet de définir les chemins de recherche des bibliothèques et des fichiers Include. Ce programme se trouve dans le répertoire C:\QC25\BIN. Dès que vous aurez entré la commande

```
C:\QC2\BIN\NEW-VARS <Return>
```

l'installation sera terminée. Vous pouvez à présent lancer le compilateur par

```
QC <Return>
```

Etant donné qu'il est nécessaire d'activer le programme NEW-VARS.BAT avant chaque lancement de Quick C 2.5, nous vous conseillons d'intégrer cette instruction à la fin dur fichier AUTO-EXEC.BAT.

1.2. L'environnement de développement

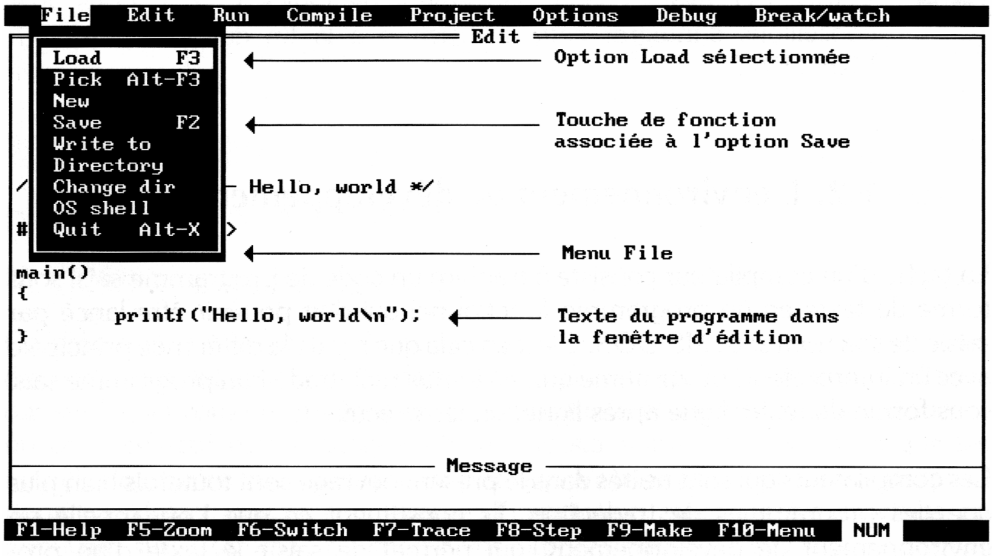
La tâche d'un compilateur consiste à traduire un code de programme saisi sous forme de texte en un programme fonctionnel qui doit pouvoir être lancé par saisie de son nom sous MS-DOS. C'est en cela que réside la différence principale avec un interpréteur, programme qui, en temps réel, traduit un programme saisi sous forme de texte, ligne après ligne, en les exécutant.

Les compilateurs qui sont traités dans le présent ouvrage sont toutefois bien plus que des programmes de traduction. Ils constituent ce que l'on appelle un environnement de développement, qui permet de saisir le texte d'un programme, de traduire et de fusionner des modules de programme, et aussi de lancer des programmes. Ils sont de plus agrémentés de débogueurs intégrés, utilitaires développés spécialement pour permettre de localiser et de corriger rapidement des erreurs.

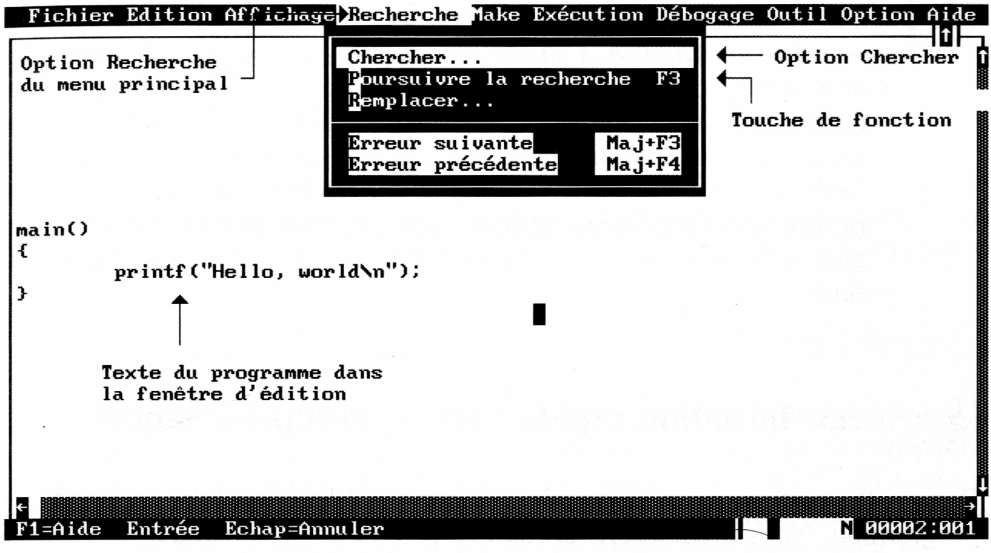
Les compilateurs renferment un système de menus bien étudié qui, par l'intermédiaire d'un menu principal, permet d'accéder à toutes les fonctions du compilateur. Quick C et C++ supportent d'autre part le travail avec la souris. Sous Turbo C et C++, c'est la touche <F10> qui vous permet d'accéder au menu principal. Dans Quick C, la touche qui assume cette fonction est <Alt>. La sélection directe d'une option de menu dans Quick C est possible par saisie de <Alt>+<Initiale de l'option désirée>. C'est ainsi que la saisie de <Alt>+<F> permet de sélectionner directement le menu *Fichier*. Celui-ci permet de charger, sauvegarder, créer ou enregistrer sous un autre nom des fichiers programme. Dans Turbo C et C++, le menu équivalent s'appelle *File*.

Grâce aux touches <Curseur gauche> et <Curseur droite> vous pouvez choisir les options du menu principal. Pour valider un choix, appuyez sur <Return>.

Dès lors qu'une option en dissimule d'autres, un menu supplémentaire s'ouvre à l'écran. Dans celui-ci, vous pourrez sélectionner l'option de votre choix à l'aide des touches <Curseur bas> et <Curseur haut>, et confirmer votre choix par une pression sur <Return>. Une option sélectionnée est affichée en inversion vidéo. <Esc> vous permet d'annuler le choix d'une option.



Le système des menus dans Turbo C 2.0



Le système des menus dans Quick C 2.5

De nombreuses options peuvent également être activées par l'intermédiaire de touches de fonction. Sous Turbo C ou C++, vous pouvez par exemple charger directement un fichier suite à une pression sur <F3>. L'expérience montre que le néophyte préfère d'abord passer par les options des menus, et évolue progressivement vers l'utilisation des touches de fonction. Dans le présent ouvrage, nous utiliserons aussi bien les touches de fonction que les options des menus.

Une autre particularité qui caractérise les environnements de développement réside dans la multiplicité des zones de travail : les fenêtres. Dans un premier temps, nous nous intéresserons à la fenêtre dans laquelle sont saisis et modifiés les textes des programmes (la fenêtre d'édition), et à la fenêtre dans laquelle sont affichés les résultats renvoyés par nos programmes (appelée User screen dans Turbo C et C++). Lorsque vous lancez votre compilateur, vous vous trouvez automatiquement dans la fenêtre d'édition. Dans Quick C 2.5, vous pouvez passer dans l'écran utilisateur à l'aide de la touche <F4>. Les compilateurs provenant de chez Borland, Turbo C et C++, utilisent dans ce but la combinaison de touches <Alt>+<F5>.

Cette rapide introduction ne suffit pas à remplacer l'étude des guides d'utilisation des compilateurs qui seront traités ici. Nous sommes toutefois en mesure d'utiliser dès à présent ces compilateurs pour atteindre les objectifs que nous nous fixons. Les tableaux récapitulatifs concernant les commandes d'édition et l'affectation des touches de fonction, que vous trouverez en annexe dans le présent ouvrage, faciliteront d'autre part votre initiation. Au fur et à mesure de votre progression dans les chapitres qui suivent, vous découvrirez de plus en plus de facettes de votre compilateur et lui porterez un intérêt grandissant.

1.3. Initiation rapide : éditer, compiler, lancer

Dans cette section, vous apprendrez comment éditer, traduire et lancer un programme. Nous modifierons ensuite ce programme, demanderons sa nouvelle traduction, le lancerons, et enfin, le sauvegarderons.

Activez le compilateur par saisie de TC <Return> (pour Turbo C et C++) ou QC <Return> (pour Quick C). Saisissez ensuite le programme dont le listing suit. Rappelons toutefois que les numéros de lignes ne doivent pas faire partie de votre saisie.

Le programme INTRO.C.

```
[1]    main()
[2]    {
[3]    printf("\nBonjour tout le monde");
[4]    }
```

A la fin de la saisie, vous voyez, dans la fenêtre d'édition de l'environnement de développement, ce que l'on nomme le "code source" de notre programme. Il s'agit du texte du programme que, grâce à l'éditeur, nous pouvons modifier, sauvegarder, au besoin à nouveau charger. Il s'agit maintenant de traduire ce code source. Dans Turbo C et C++, appuyez à cet effet sur les touches <Alt>+<C>, <M>, et dans Quick C, activez l'option *Make/Compiler*. Nous lançons ainsi la traduction du code source en un programme exécutable. Dès que le code source a été traduit et qu'aucune erreur n'y a été décelée (dans cet exemple, vous pouvez ne pas tenir compte d'éventuels messages d'erreur), nous pouvons lancer le programme par <Ctrl>+<F9> (dans Turbo C et C++) ou <F5> (dans Quick C).

Pour voir ce que renvoie le programme, il nous faut maintenant passer dans l'écran utilisateur par pression sur <Alt>+<F5> dans Turbo C et C++, ou sur <F4> dans Quick C. Vous voyez sur votre écran apparaître l'expression

```
Bonjour tout le monde
```

C'est là le résultat renvoyé par notre programme. Appuyez sur une touche quelconque pour revenir à la fenêtre d'édition. Déplacez le curseur dans la ligne 3, et complétez la ligne de programme de manière à obtenir l'expression suivante :

```
printf("\nBonjour tout le monde, ici C");
```

A présent, lancez directement le programme ainsi modifié par <Ctrl>+<F9> ou, sous Quick C, en appuyant sur <F5>. Vous constatez que le compilateur traduit d'abord le programme, puisque son code source a été modifié, et qu'il ne le lance qu'ensuite. Passez dans la fenêtre utilisateur (respectivement <Alt>+<F5> ou <F4>). Le nouveau résultat renvoyé est maintenant :

```
Bonjour tout le monde, ici C
```

Vous voyez que le résultat retourné par notre programme de démonstration a changé. Vous venez d'écrire votre premier programme en C, et nous allons le sauvegarder. Dans ce but, sous Turbo C ou C++, vous appuierez sur la touche <F2>. Dans Quick C, activez l'option *Fichier/Enregistrer* <Alt>+<F>+<E>.

Nous venons donc de créer, modifier, traduire, lancer et enfin sauvegarder un programme. Ce sont là les étapes élémentaires de la programmation. Avant de nous lancer dans l'apprentissage systématique du C dans le chapitre 2, voyons encore quelques notions fondamentales qui sont à la base de l'apprentissage d'un langage de programmation.

Un programme est destiné à traiter des informations. Ces informations sont appelées "données". Ces données doivent d'abord être entrées, le programme les traite alors, puis renvoie le résultat de son traitement à l'utilisateur. Appelons ETS le principe d'une telle procédure fonctionnant par Entrée, Traitement, Sortie. Lorsque vous désirez écrire un programme, il vous faut connaître les commandes qui permettent de mettre en application ce principe dans le langage de programmation que vous utilisez.

Un traitement peut consister en l'exécution de plusieurs commandes successives (exécution séquentielle), mais il peut aussi être nécessaire que certaines

commandes soient répétées en fonction de certaines conditions (boucles), ou qu'encore une commande ne doive être exécutée que sous certaines conditions (renvois). De par sa structure, un programme pourra donc contenir des séquences d'instructions successives, des boucles et des renvois. Etant donné que ces structures contrôlent et gèrent l'exécution du programme, on les appelle habituellement "structures de contrôle". Lorsque vous désirez apprendre un langage de programmation, il faut d'abord savoir comment programmer ces structures de contrôle.

L'on appelle "données" des informations qu'un ordinateur est capable d'interpréter et de traiter. Tout programmeur doit s'interroger sur la nature des données qui permettent de résoudre un problème, et les données qui sont nécessaires à l'élaboration d'une solution. En plus de la structure d'un programme, les données constituent donc le second sous-ensemble fondamental d'un programme.

Dans les langages de programmation évolués tels que C, Pascal, COBOL, PL1 ou MODULA2, une déclaration convenable des données à traiter est essentielle. L'ordinateur doit en effet savoir s'il doit travailler sur un nombre ou sur une chaîne quelconque de caractères. Les données devant être mémorisées, il est nécessaire de leur réserver une plage de la mémoire de travail. C'est pourquoi il est important de définir la nature d'une valeur numérique : il faut spécifier si un nombre est entier, s'il est muni d'un signe, ou s'il comporte une virgule flottante.

Des structures de données différentes décrivent des types de données distincts. Seules les définitions de la structure des données et de la structure d'un programme permettent de résoudre des problèmes de programmation. Dans la suite de ce livre, vous apprendrez par conséquent à définir ces deux unités fondamentales dans le langage de programmation C.

Les programmes volumineux ne sont généralement pas rédigés sous forme d'une entité unique, mais plus souvent divisés en une partie assurant la gestion, couramment appelé "programme principal", et en plusieurs sous-programmes. Un sous-programme exécute généralement une tâche bien spécifique, et est activé par d'autres parties du programme. En C, les sous-programmes sont généralement qualifiés de "fonctions".

2 Premiers pas en C

Dans ce chapitre, nous continuerons d'expliquer le travail dans les environnements de développement qui font l'objet de ce livre, en nous basant sur d'autres exemples de programmes. Vous allez charger, éditer et compiler un programme. Par l'intermédiaire d'erreurs volontaires courantes, vous vous initierez à la correction d'erreurs. Vous apprendrez également à interpréter correctement les messages d'erreurs retournés par les compilateurs. Les programmes de démonstration qui seront traités dans ce chapitre simulent, malgré un travail minime, un programme C complet et capable de fonctionner. L'accent est mis ici sur la programmation du renvoi de texte à l'écran.

Avant de vous lancer dans l'utilisation d'un langage de programmation, il faut que vous sachiez quel nom vous pouvez donner aux éléments de votre programme (les variables, les fonctions, les définitions de données etc.).

Dans le langage C, un nom est une combinaison de caractères alphanumériques, dans laquelle le premier caractère doit être une lettre quelconque de l'alphabet ou le tiret de soulignement.

Le compilateur Turbo C supporte également le symbole du dollar. Réfléchissez toutefois bien avant de l'employer, car l'usage de ce symbole n'est pas très répandu auprès des programmeurs en C, la plupart des compilateurs ne l'acceptant pas. Si par conséquent vous désirez écrire des programmes portables, il est préférable que vous n'utilisiez pas le signe du dollar.

Pour ce qui concerne l'attribution de noms en C, il convient de se conformer à trois règles :

① Il faut faire la distinction entre majuscules et minuscules. Le nom *INDEX* désignerait une autre variable que le nom *index*, et aucune de ces deux variables ne serait identique à celle qui porte le nom *inDex*. Bref : à orthographe différente, variables différentes !

② Les 32 premiers caractères d'un nom sont significatifs. Dès que vous utilisez plus de 32 caractères, le compilateur ignore ceux qui sont situés au-delà du trente deuxième.

③ N'utilisez pas de caractères accentués ni de cédille.

Bien que le tiret de soulignement puisse contribuer à la lisibilité d'un code de programme, son usage semble peu répandu auprès des programmeurs en langage C. Alors que le programmeur en Pascal affectionne le tiret de soulignement pour écrire des noms très explicites, les programmeurs en C ont tendance à choisir des désignations courtes et codées.

2.1. Le plus petit programme C possible

Pour faire nos premiers pas dans le langage de programmation C, le meilleur moyen consiste à observer un programme écrit en C.

Listing 1 : Le programme RIEN.C.

```
[1 ]    main()
[2 ]    {
[3 ]    }
```

Vous voyez ci-dessus le programme C le plus simple que l'on puisse imaginer. Impossible d'en retirer quelque chose ou d'y apporter une modification. Malheureusement, ce programme ne fait absolument rien.

Le minimum qu'exige un programme C est de contenir la fonction *main()*. Cette fonction doit obligatoirement être appelée dans chaque programme C. *Main()* est l'en-tête de tout programme C.

L'appel de la fonction *main()* est suivi d'une accolade ouvrante et d'une accolade fermante, caractéristiques qui signalent au compilateur que le nom *main()*

désigne une fonction. Dans le chapitre 6, nous verrons exactement ce que l'on entend par "fonction" dans C.

Les deux accolades définissent le début et la fin de *main()*. Lorsque vous lancez le programme, toutes les instructions écrites entre les accolades seront exécutées les unes après les autres. Suite à cela, l'exécution du programme sera achevée.

Lancez le programme par <Ctrl>+<F9> ou, sous Quick C, par pression sur <F5>, puis passez dans l'écran utilisateur par <Alt>+<F5> (respectivement <F4>). Vous ne verrez rien sur l'écran de sortie. Pourtant, vous avez sous les yeux un programme C complet exempt d'erreurs.

2.2. Un programme avec sortie sur l'écran

Passons à un exemple plus intéressant. Lorsque vous entrez ce programme, veillez à ne pas saisir les numéros des lignes. Lors de la sauvegarde qui suivra, veuillez utiliser le nom proposé dans le listing.

Listing 2 : Le programme AGIS.C

```
[1 ]  /* Chapitre 2.2 */
[2 ]
[3 ]  /* Nom du programme : AGIS.C */
[4 ]  /* Exemple de renvoi de texte simple */
[5 ]
[6 ]  main()
[7 ]  {
[8 ]    printf("Ceci est un renvoi d'une ligne de texte sur l'écran.");
[9 ]  }
```

Il s'agit presque du même programme que le précédent. La différence réside dans le fait que celui-ci renferme une instruction exécutable entre les accolades ! Cette instruction *printf* consiste en l'activation d'une fonction que le compilateur met à votre disposition.

La fonction *printf()* fait en sorte que la chaîne de caractères (en anglais : string) écrite entre guillemets dans les parenthèses soit renvoyée à l'écran à partir de la position courante du curseur. Par conséquent, tout ce qui figurera dans les parenthèses de cette fonction, entre les guillemets, sera renvoyé sur l'écran par la fonction *printf()*.

Notez bien la présence du point virgule à l'extrémité de la ligne 8 !

En C, le point virgule symbolise la fin d'une commande.

Chaque ligne de programme qui se termine par un point virgule est appelée "instruction".

Compilez et lancez à présent ce programme par action sur <Ctrl>+<F9> ou <F5> (Quick C), et observez ce qui se produit.

Voici l'affichage retourné par le programme AGIS.C :

```
Ceci est un renvoi d'une ligne de texte sur l'écran.
```

Ce listing présente une autre nouveauté : le commentaire qui s'étend de la ligne 1 à la ligne 4 est délimité par /* et */. Mais soyez sans crainte : la technique d'intégration de commentaires dans un programme sera étudiée en détail dans un chapitre ultérieur.

2.3. Travailler avec le compilateur

Avant de découvrir des programmes supplémentaires plus volumineux dans les chapitres qui vont suivre, nous allons nous consacrer à certaines fonctions de base nécessaires au travail avec un compilateur C. A cet effet, nous allons modifier le programme AGIS.C à l'aide des compilateurs Turbo C 2.0, C++ 3.0 et Quick C 2.5. Ensuite, nous le ferons traduire et le lancerons depuis l'environnement de développement. Etant donné que les différences entre les compilateurs sont minimes, nous représenterons toutes les fonctions en nous basant sur l'exemple du Turbo C. Lorsqu'apparaîtront des différences par rapport à C++ et à Quick C (d'autres options ou commandes d'édition, par exemple), celles-ci seront mentionnées entre parenthèses derrière les options et commandes du Turbo C.

2.3.1. Edition de programmes C

En informatique, le verbe "éditer" désigne la saisie et la correction de textes. Dans l'environnement de développement du C, les programmes C sont saisis ou corrigés à l'aide du programme de saisie de texte intégré à l'environnement, couramment appelé "Editeur". Ensuite, ces programmes sont enregistrés sous forme de fichiers ASCII. Les fichiers programme possèdent toujours l'extension ".C". Des codes programme déjà enregistrés devront d'abord être chargés dans le module d'édition de l'environnement de développement dès lors que l'on désirera les corriger ou les développer. A cet effet, appuyez sur la touche <F3> puis <Return>, ou, sous Quick C, sélectionnez l'option *Fichier/Charger*. Le compilateur vous montre alors une liste de tous les fichiers programme C dans laquelle vous pourrez sélectionner le fichier désiré à l'aide des touches de déplacement, ou en cliquant sur le nom approprié avec la souris.

Maintenant que nous avons chargé le programme AGIS.C, nous nous proposons d'apporter quelques modifications à son texte. A l'aide des touches de déplacement, placez le curseur dans la ligne 6. Appuyez ensuite sur la combinaison de touches <Ctrl>+<Y>. La ligne 6 disparaît. Rendez-vous dans la ligne 7, et appuyez sur la touche <End>. Le curseur se trouve à présent derrière le point virgule qui conclut la ligne. Effacez ce point virgule en appuyant sur la touche <Backspace>.

Voyons maintenant les effets produits par nos modifications lorsque nous demanderons au compilateur de traduire le nouveau code du programme.

2.3.2. Compilation de programmes C

Activez la traduction de notre code programme en un programme fonctionnel par la succession de touches <Alt> <C> puis <C>. Sous Quick C, sélectionnez à cet effet l'option *Make/Compiler*. Vous constatez que le compilateur signale une erreur, et qu'il positionne le curseur dans la ligne où se trouve l'accolade ouvrante. C'est à cet endroit du code du programme que le compilateur a détecté une erreur de syntaxe. Une erreur de syntaxe est le non respect d'une règle du langage de programmation C. Cela ne signifie pas nécessairement que l'erreur se trouve effectivement dans cette ligne. La seule chose que nous puissions affirmer, c'est qu'elle se trouve soit dans cette ligne soit dans celles qui précèdent.

Si vous avez retenu ce dont il était question dans la section précédente, la règle qui n'a pas été respectée doit vous sembler évidente : il manque l'en-tête *main()*.

Créez une ligne vide en appuyant sur <Return>. Entrez dans cette ligne la chaîne de caractères *main()*, puis compilez le fichier par <Alt> <C> puis <M>, ou, sous Quick C, par l'intermédiaire de *Make/Compiler*. Une nouvelle erreur vous est signalée. Cette fois, le curseur se trouve derrière l'accolade fermante, et le compilateur signale qu'il manque un point virgule.

Voici les messages renvoyés par le compilateur lorsque manque un point virgule :

```
Statement missing ; in function main
(Turbo C, C++)
Erreur de syntaxe - ";" manquant avant '}
(Quick C)
```

Une fois de plus, l'erreur ne se trouve pas dans la ligne où figure le curseur, mais plus haut dans le programme ! Entrez le point virgule en guise de dernier caractère dans la ligne 8, et compilez le programme pour la troisième fois. Les compilateurs Turbo C et C++ signalent une traduction exempte d'erreurs, et par conséquent couronnée de succès, et vous demandent de revenir dans la fenêtre d'édition en appuyant sur une touche quelconque.

```
File   Edit   Run   Compile   Project   Options   Debug   Break/watch
                              Edit
     Line 8     Col 69  Insert Indent Tab Fill Unindent * C:AGIS.C
/* Chapitre 2.2

Nom du programme : AGIS.C
Exemple de renvoi de texte simple */
                          Compiling
main()
{                  Main file: AGIS.C
     printf("Ceci  Compiling: EDITOR → AGIS.C                  écran.");
}
                                   Total     File
                     Lines compiled: 9         9
                           Warnings: 0         0
                             Errors: 0         0

                   Available memory: 192K
                   Success         :    Press any key
                              Message
 Compiling C:\TC\AGIS.C:
•Error C:\TC\AGIS.C 9: Statement missing ; in function main
F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu  NUM
```

Turbo C à la suite d'une compilation réussie

Quick C ne renvoie aucun commentaire. Nous savons à présent que le code de notre programme est correct, et qu'il a pu être traduit en un programme susceptible de fonctionner. Ce que nous ne savons pas, par contre, ce sont les effets produits par ce programme. Pour en savoir davantage, il nous faut le lancer.

2.3.3. Lancement de programmes

L'un des aspects les plus agréables des environnements de développement réside dans le fait que, pour lancer des programmes, il n'est pas nécessaire de quitter l'environnement de développement. Dès que le code d'un programme a été modifié, le lancement s'accompagne systématiquement d'une nouvelle compilation. Sous Turbo C et C++, vous lancerez un programme par la combinaison de touches <Ctrl>+<F9>, et en Quick C par pression sur <F5>.

Quick C passe dans la fenêtre utilisateur, dans laquelle vous pouvez suivre l'exécution du programme activé. Une pression sur une touche quelconque vous amène ensuite dans la fenêtre d'édition. Dans Turbo C 2.0 et C++, il faut passer dans la fenêtre utilisateur en actionnant la combinaison de touches <Alt>+<F5>.

2.3.4. Correction d'erreurs

Dans les paragraphes qui suivent, nous allons infiltrer deux erreurs typiques supplémentaires dans notre programme de démonstration. Celles-ci nous aideront à préciser le principe de fonctionnement du compilateur, et nous fourniront des renseignements importants dans l'optique d'une correction d'erreurs efficace. A cet effet, positionnez le curseur dans la ligne 8 sur les guillemets qui concluent la chaîne alphanumérique. Effacez ce caractère par pression sur <Del>, puis lancez le programme ainsi modifié.

Le compilateur traduit d'abord le texte du programme et, à notre grande surprise, signale deux (!) erreurs. La première erreur réside dans l'absence de guillemets pour conclure la chaîne de caractères intégrée à la fonction *printf()*. La deuxième erreur est une conséquence de la précédente : le compilateur prétend qu'il manque une parenthèse fermante.

Quelle est la cause de ce deuxième message d'erreur ? En fait, le compilateur considère que, dans la ligne

```
printf("Ceci est un renvoi d'une ligne de texte sur l'écran.);
```

la chaîne doit être terminée par des guillemets. C'est pourquoi, à ses yeux, la parenthèse fermante n'est pas celle qui achève la fonction, mais une partie de la chaîne alphanumérique inachevée.

Une opération unique suffit à corriger les deux erreurs : il suffit d'insérer les guillemets en fin de chaîne. Lorsque vous aurez apporté cette correction, amenez le curseur dans la ligne 7, et effacez-y l'accolade ouvrante. Lancez alors le programme. Durant la compilation, le compilateur de Turbo C signale la présence de deux erreurs de syntaxe. Quick C n'en mentionne qu'une. Tous deux spécifient que l'instruction *printf()* leur est inconnue. A la suite de *main()*, il est donc impératif de placer l'accolade ouvrante. Si celle-ci n'est pas présente, des lignes de programme pourtant correctes seront considérées comme étant erronées.

Notre petit programme de démonstration nous permet de voir qu'il existe un grand nombre de sources d'erreurs, et il est essentiel de savoir interpréter correctement les messages d'erreur renvoyés par le compilateur utilisé pour pouvoir apporter au texte du programme les corrections appropriées. Nous avons vu qu'il est tout à fait possible qu'une erreur unique engendre plusieurs erreurs de second ordre. Dans le chapitre 16.3, vous trouverez une récapitulation des erreurs de programmation les plus fréquentes en C.

2.4. Programmation de renvois de texte

Le programme qui suit sera notre premier exemple mettant en évidence la gestion de l'affichage de texte. Grâce à ce programme, vous verrez comment programmer une rupture de ligne suivie d'un saut de ligne. Entrez le programme CONTINU.C qui, dans *main()*, contient les instructions suivantes.

Listing 3 : Le programme CONTINU.C

```
[1 ]  main()
[2 ]  {
[3 ]    printf("Ceci est une ligne de texte suivie d'un saut de ligne.\n")
[4 ]    printf("En voici une autre, à laquelle -->");
[5 ]    printf("ce texte est rattaché.\n\n");
[6 ]    printf("Ceci est la troisième ligne de texte, qui suit une ligne
            vide.\n");
[7 ]  }
```

Comme vous pouvez le constater, nous avons à faire ici à quatre instructions dont chacune est introduite par l'activation de la fonction *printf()*. Ces quatre instructions sont traitées successivement, dans leur ordre chronologique. Un tel principe de fonctionnement est qualifié de traitement séquentiel ou linéaire.

A la fin de la première ligne de commandes, vous constatez la présence d'un backslash ("\"). Dans la fonction *printf()*, ce caractère est utilisé pour faire savoir au compilateur qu'à la suite de ce signe se trouve un caractère de contrôle particulier : une séquence d'échappement (ou séquence escape). Dans notre cas, la lettre "n" qui suit le backslash signifie qu'à la suite de l'affichage de la chaîne de caractères il y aura un saut de ligne. Suite à cela, le curseur sera positionné à l'extrême gauche de la ligne suivante. On parle couramment dans ce cas d'un "retour chariot" (en anglais : Carriage Return ou CR) suivi d'un saut de ligne (en anglais : Line Feed ou LF).

Le code de contrôle *n* peut être placé à des endroits quelconques, autant de fois que vous le désirez, dans une chaîne de caractères à visualiser. Vous pouvez ainsi créer vous-même des césures de mots. Le compilateur interprète systématiquement le backslash suivi d'un caractère quelconque comme un code de contrôle. Si vous désirez utiliser le backslash comme caractère dans un texte, il faudra y faire figurer la chaîne de caractères "\\".

Ces précisions étant apportées, nous pouvons maintenant décrire le programme précédent. La première ligne *printf* renvoie une chaîne de caractères à l'écran et positionne le curseur dans la ligne suivante. La chaîne de caractères à retourner est délimitée par des guillemets dans la parenthèse associée à la fonction. Le saut de ligne est déclenché par la séquence d'échappement \n. La deuxième instruction retourne, elle aussi, un texte, mais cette fois sans saut de ligne. Le texte qui suit, intégré dans la troisième instruction *printf*, sera par conséquent écrit immédiatement à la suite du texte précédent, dans la même ligne. Suite à cela, l'instruction impose à nouveau un saut de ligne suivi d'une ligne vide occasionnée par le deuxième code de contrôle *n*. Pour finir, c'est le texte associé à la quatrième et dernière instruction *printf* qui est affiché à l'écran.

Compilez et lancez le programme par <Ctrl>+<F9> ou, sous Quick C, par pression sur <F5>.

Voici l'affichage produit par le programme CONTINU.C

```
Ceci est une ligne de texte suivie d'un saut de ligne.
En voici une autre, à laquelle --> ce texte est rattaché.
Ceci est la troisième ligne de texte, qui suit une ligne vide.
```

A titre d'exercice, vous pouvez intégrer à ce programme d'autres instructions d'affichage. (Pour faire vos propres exercices, utilisez toujours une copie du programme de démonstration original !).

Chargez maintenant le programme AGIS.C. Remplacez-y l'instruction *printf()* par la ligne suivante :

```
printf("H\tW\nE\tO\nL\tR\nL\tL\nO\tD\n");
```

Sauvegardez ce fichier par <F10>+<F>+<A> (en C++), ou <F10>+<F>+<W> (en Turbo C), ou encore <Alt>+<F>+<S> (sous Quick C) en lui donnant le nom HELLO1.C.

Dans cette instruction, nous venons d'utiliser une nouvelle séquence d'échappement : *t*. Cette séquence de contrôle fait en sorte que le curseur soit déplacé vers la droite d'un saut de tabulation avant d'afficher le texte.

Vous constatez que, de même que le code de contrôle *n* (= nouvelle ligne), la séquence de contrôle *t* (= saut de tabulation) peut être placée autant de fois que vous le désirez, et aux emplacements de votre choix, dans une chaîne alphanumérique.

Compilez et lancez le programme.

```
H       W
E       O
L       R
L       L
O       D
```

Cet exemple vous prouve que, pour obtenir un affichage surprenant pour l'oeil, il n'est pas nécessaire de produire d'énormes efforts de programmation.

2.5. Renvoi de nombres

Voici un exemple de traitement de données dans un programme C.

Listing 4 : Le programme UNNOMBRE.C

```
[1 ]  /* Chapitre 2.4 */
[2 ]
[3 ]  /* Nom du programme : UNNOMBRE.C */
[4 ]  /* Exemple de renvoi d'un nombre décimal */
[5 ]
[6 ]
[7 ]  main()
[8 ]  {
[9 ]
[10]     int index
[11]     int var1;
[12]     int var2;
[13]
[14]     index = 13;
[15]     var1 = 27;
[16]     var2 = 30;
[17]
[18]     printf("La valeur de index est %d, celle de var1 est %d et
[19]            celle de var3 est %d\n",index, var1, var2);
[20]
[21] }
```

L'en-tête *main()* ainsi que l'accolade ouvrante qui le suit, ne devraient plus avoir de secrets pour vous. L'innovation, dans cet exemple, se trouve à la ligne 10 :

```
int index;
```

Celle-ci sert à déclarer une variable entière dont le nom est *index*. Le mot *int* est un mot clé du C.

Les mots clé font partie intégrante d'un langage de programmation, dans lequel ils ont une signification prédéfinie. C'est pourquoi ils ne peuvent être utilisés comme dénominations d'autres éléments.

Vous trouverez en annexe un récapitulatif des mots clé normalisés. Le mot clé *int* définit une variable de type entier dont la valeur peut être comprise entre -32768 et 32767.

Dans un programme, les variables sont par conséquent des éléments de substitution dont le contenu peut varier pendant l'exécution d'un programme.

Le nom de variable qui suit *int*, soit *index* dans notre cas, peut être quelconque, à condition qu'il respecte les règles d'attribution de noms spécifiques au C, et que son orthographe ne soit pas la même que celle d'un mot clé réservé de ce langage.

Notez que nous n'avons encore attribué aucune valeur à la variable *index* définie ci-dessus. Nous nous sommes contentés de spécifier que la variable dont le nom est *index* sera une variable entière. Dans la suite de ce livre, nous désignerons par "déclaration" un procédé de ce genre.

Dans les deux lignes qui suivent, nous déclarons deux autres variables : *var1* et *var2*. Dans les lignes 14 à 16 vous voyez de quelle façon une valeur est attribuée aux variables index, *var1* et *var2*. Dans la ligne 18, le contenu (ou, si vous préférez, la valeur affectée) des trois variables est renvoyé sur l'écran.

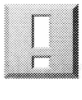
Voici l'affichage que produit le programme UNNOMBRE.C :

```
La valeur de index est 13, celle de var1 est 27 et celle de var2 est 30
```

Voyons comment cet affichage a été programmé. Si vous vous remémorez nos programmes de démonstration précédents, vous pouvez constater que certaines différences distinguent l'instruction *printf* de l'exemple courant de celles des programmes précédents. La première différence évidente concerne le caractère %. Ce caractère signale à la fonction *printf()* qu'à l'endroit qu'il occupe il convient d'interrompre l'affichage de la chaîne de caractères en cours, et d'écrire le contenu d'une variable.

Le caractère % est utilisé pour définir le format dans lequel le contenu de variables devra être affiché.

Dans notre exemple, le type de variable est spécifié par la lettre *d* qui suit le symbole %. Cela signifie qu'à cet emplacement de la chaîne de caractères le programme doit affiché une valeur entière. La chaîne de caractères *%d* est appelée "descripteur de champ" (en anglais : field descriptor). Etant donné que sont renvoyés à l'écran trois contenus de variables, il faut que, dans la chaîne associée à l'instruction *printf* figurent trois descripteurs de champs. A la suite du dernier, nous avons placé la séquence d'échappement qui génère un saut de ligne.



Nous sommes maintenant en mesure de comprendre le format général de la fonction

```
printf("String de mise en forme contenant des séquences d'échappement
      et des descripteurs de champs", liste de variables);
```

Tous les caractères écrits entre guillemets définissent le format de l'affichage (également appelé masque) des données que la commande *printf* est chargée de renvoyer à l'écran. Suivent ensuite, séparés par des virgules, les noms des variables à insérer dans le masque : la liste des variables.

Dans une instruction de *printf*, vous pouvez intégrer un nombre quelconque de descripteurs de champs et de variables. Une caractéristique est toutefois importante : le nombre des descripteurs de champs figurant dans la chaîne de mise en forme doit être égal à celui des variables énumérées dans la liste des variables, en respectant le même ordre. La première variable à intégrer à la chaîne de mise en forme doit aussi être la première dans la liste des variables, et ainsi de suite.

Nous aborderons plus loin tous les aspects de la mise en forme des entrées/sorties en C. Pour l'instant, les connaissances de base acquises jusque-là suffiront pour les chapitres à venir.

2.6. Intégration de commentaires dans des programmes C

Le programme qui suit est un exemple d'intégration de commentaires dans un programme C. Les commentaires servent à améliorer la lisibilité d'un programme, afin de pouvoir s'y retrouver par la suite. La chaîne de caractères /* signale le début d'un commentaire, et */ symbolise la fin d'un commentaire. Le texte compris entre ces deux délimiteurs est interprété comme commentaire par le compilateur, et n'est pas compilé.

Listing 5 : Le programme COMMENTE.C

```
[1 ]  /* Chapitre 2.6 */
[2 ]
[3 ]  /* Nom du programme : COMMENTE.C */
[4 ]  /* Exemple de programme commenté */
[5 ]
[6 ]  main()                        /* Ceci est l'en-tête du programme*/
[7 ]  {    /* Début de MAIN */
[8 ]   printf("Voyons comment on intègre des "); /* Ligne_MILIEU renvoyée */
[9 ]  printf("commentaires dans des programmes C")/* ligne supplémentaire:*/
[10]                              /* ce commentaire s'étend sur deux lignes */
[11]  }    /* Fin de MAIN */
```

Cet exemple vous montre qu'un commentaire peut aussi s'étendre sur plusieurs lignes. Le style de programmation actuellement le plus prisé consiste à intégrer une description succincte du programme, au début du texte de celui-ci, dans des lignes de commentaires. Les commentaires peuvent être insérés à tout emplacement dans le programme.

Les commentaires sont extrêmement importants dans tous les langages de programmation. Il n'est en effet pas rare qu'un programmeur oublie pourquoi une procédure donnée a été programmée d'une façon plutôt que d'une autre. Il peut donc intégrer des mentions appropriées dans le texte d'un programme qui, par la suite, lui permettront de retrouver le fil de son raisonnement. Nous reviendrons d'ailleurs sur le style de la programmation dans le chapitre 17.

2.7. Récapitulation

Chaque programme C doit contenir la fonction *main()*. Lorsque l'on lance un programme, toutes les instructions contenues dans *main()* sont exécutées dans leur ordre chronologique. Lorsque la dernière instruction a été exécutée, le programme est terminé.

Les variables doivent être déclarées. Cela signifie que, pour chaque variable, il faut spécifier le type de données dont il s'agit, ainsi que le nom de la variable. La déclaration des variables doit figurer dans le programme avant leur utilisation.

La fonction *printf()* sert à renvoyer des chaînes de caractères sur l'écran. Grâce aux descripteurs de champs, des contenus de variables peuvent être intégrés à la chaîne de caractères à afficher. Le descripteur de champ *%d* retourne le contenu de variables sous forme de valeur entière, et peut être utilisé pour afficher des données dont le type est *int*.

Des commentaires peuvent être intégrés à tout emplacement dans un programme C. Les délimiteurs de commentaires sont les chaînes /* et */, correspondant respectivement au début et à la fin d'un commentaire.

Exercices se rapportant au chapitre 2

Exercice 1 :
Ecrivez un programme qui renvoie votre nom à l'écran. Donnez lui le nom EXO1.C.

Exercice 2 :
Modifiez le programme EXO1.C de façon à ce que soient également renvoyés à l'écran votre adresse.

Exercice 3 :
Ecrivez un programme qui renvoie sur l'écran votre nom de famille et votre prénom, verticalement, séparés par un espace de deux sauts de tabulation. Utilisez à cet effet une instruction printf unique. La forme de l'affichage obtenu devrait être du genre suivant :

```
P        J
E        E
T        R
I        O
T        M
J        E
E
A
N
```

3 Structures de contrôle en C

Ce chapitre vous fournit une vue d'ensemble complète de la programmation des structures de contrôle. Sur la base d'exemples de programmes, vous découvrirez la programmation des boucles à cycle facultatif (testées en entrée) et des boucles à cycle obligatoire (testées en sortie), ainsi que les renvois conditionnels et inconditionnels, et la structure des différenciations de cas. Par la même occasion, vous apprendrez des techniques supplémentaires de programmation de l'affichage à l'aide de la fonction *printf()*.

Les structures de contrôle définissent l'ordre chronologique dans lequel doivent être traitées les instructions contenues dans un programme. Etant donné que, implicitement, le traitement est linéaire (ou séquentiel), c'est-à-dire que les instructions sont traitées dans l'ordre dans lequel elles apparaissent dans le texte du programme, chaque exception doit être programmée à l'aide de mots clé et d'instructions particulières.

Le langage de programmation C connaît plusieurs types de constructions pour programmer des boucles (en anglais : loop) et des renvois conditionnels (en anglais : conditional branching).

3.1. Boucles testées en entrée à l'aide de while

Dans une boucle *while*, les instructions sont répétées aussi longtemps qu'une condition définie à cet effet est remplie. Dès l'instant où la condition n'est plus remplie, la boucle s'interrompt, et l'exécution du programme se poursuit par le traitement de l'instruction qui suit la boucle.

Le programme WHILE.C nous servira d'introduction. Dans ce programme, de même que dans ceux qui suivront, nous commenterons chaque ligne de programmation, de manière à ce qu'à tout instant l'on sache pourquoi une ligne a été programmée.

Listing 6 : Le programme WHILE.C

```
[1 ]  /* Chapitre 3.1 */
[2 ]
[3 ]  /* Nom du programme : WHILE.C */
[4 ]  /* Ceci est un exemple de boucle à cycle facultatif utilisant WHILE */
[5 ]
[6 ]  main()
[7 ]  { /* Début de MAIN */
[8 ]    int compteur;              /* Variable de type entier Compteur */
[9 ]
[10]    compteur = 0;              /* Variable Compteur est mise à 0 * /
[11]
[12]    while (compteur < 6)       /* tant que compteur est inférieur à 6 */
[13]    { /* Début de WHILE */
[14]      printf("La valeur de la variable incrémentée est %d\n",compteur);
                                   /* Affichage */
[15]      compteur = compteur + 1; /* Variable incrémentée de 1 */
[16]    } /* Fin de WHILE */
[17]
[18]  } /* Fin de MAIN */
```

Après plusieurs lignes de commentaires, le programme commence réellement dans la ligne 6. Ensuite, dans la ligne 8, est déclarée la variable entière *compteur*. Dans la ligne 10, la valeur 0 est affectée à cette variable avant (!) le début de la boucle, qui, elle, s'étend sur les lignes 12 à 16.

La boucle proprement dite est introduite par le mot clé *while*, suivi d'une expression entre parenthèses. Viennent ensuite une série d'instructions délimitées par des accolades (si la boucle ne renferme qu'une seule instruction, vous pouvez vous dispenser des accolades !). Les accolades signalent ici le début et la fin de l'ensemble des instructions qui composent la boucle, c'est-à-dire qu'elles délimitent le corps de la boucle.

Aussi longtemps que la condition figurant dans les parenthèses de *while* est remplie, les instructions composant le corps de boucle seront exécutées. Dans notre cas, la variable *compteur* sera incrémentée de 1 à chaque cycle, cette variable atteignant ainsi la valeur 6 au bout de six cycles. A cet instant, la condition d'exécution de la boucle n'est plus remplie, et l'exécution du programme se poursuit après le corps de boucle. Une variable qui assume la fonction de la variable *compteur* dans notre exemple est également appelée variable de boucle, ou variable d'incrémentation.

Dans la section qui suivra, nous reviendrons plus en détail sur l'expression

```
(compteur < 6)
```

mentionnée entre parenthèses. Dans l'immédiat, évitez de vous poser des questions sur les effets produits par ces parenthèses.

Notez les particularités suivantes concernant la boucle *while* de l'exemple ci-dessus :

Si la variable *compteur* avait été fixée au départ à une valeur supérieure à 5, la boucle n'aurait pas été parcourue une seule fois.

Si, dans le corps de boucle, la valeur de la variable *compteur* n'augmentait pas, la boucle serait infinie.

Compilez le programme, puis lancez-le.

L'affichage renvoyé par le programme WHILE.C est le suivant :

```
La valeur de la variable incrémentée est 0
La valeur de la variable incrémentée est 1
La valeur de la variable incrémentée est 2
La valeur de la variable incrémentée est 3
La valeur de la variable incrémentée est 4
La valeur de la variable incrémentée est 5
```

Vous pouvez constater que les instructions du corps de boucle sont exécutées six fois. Modifiez la valeur de *compteur* dans la ligne 12, en remplaçant 6 par 10. Lancez à nouveau le programme. Vous voyez que, cette fois, la boucle est parcourue dix fois de suite.

3.2. Boucles testées en sortie à l'aide de do..while

Le programme DOWHILE.C est une variante de la boucle construite sur le principe *while* : la boucle *do..while*.

Listing 7 : Le programme DOWHILE.C

```
[1 ]  /* Chapitre 3.2 */
[2 ]
[3 ]  /* Nom du programme : DOWHILE.C */
[4 ]  /* Ceci est un exemple de boucle DO..WHILE à cycle imposé */
[5 ]
[6 ]  main()
[7 ]  { /* Début de MAIN */
[8 ]     int i;
[9 ]
[10]     i = 0;
[11]     do                    /* répète tant que */
[12]     {
[13]          printf("La valeur de i est à présent %d\n",i);
[14]          i = i + 1;
[15]     } while (i < 6);      /* i est inférieur à 6 */
[16]  } /* Fin de main */
```

Ce programme est presque identique à celui que nous avons traité dans la section précédente. La différence réside dans le fait que la boucle est ici introduite, en ligne 12, par le mot clé *do*, et que le corps de boucle est suivi du mot clé *while*, lui-même suivi de la condition de répétition "i inférieur à 6", qui signale la fin de la boucle. Dans ce cas également, la boucle sera parcourue aussi longtemps que la condition définie dans les parenthèses de *while* sera remplie.

Mais alors, en quoi réside la différence ? Elle tient au fait que la construction *do..while* est parcourue au moins une fois, contrairement à la boucle *while*. Dans le présent exemple, ce n'est qu'en fin de boucle, dans la ligne 15, que le programme vérifie si la condition d'exécution de boucle est remplie. C'est pourquoi l'on parle ici d'une boucle à cycle imposé, ou testée en sortie. Une boucle *while*, par contre, est à cycle facultatif, ou testée en entrée. Dans cette dernière, le programme vérifie si la condition d'exécution de boucle est remplie avant (!) d'exécuter la boucle.

Les boucles peuvent être imbriquées à volonté. Cela signifie qu'une boucle peut en inclure une autre, contenant ses propres conditions, qui elle-même pourra en inclure une, etc. Il n'y a aucune limite quantitative à l'imbrication de boucles.

3.3. Boucles testées en entrée à l'aide de for

Une boucle *for* n'est rien de plus que la nouvelle description d'une boucle *while*. Le programme FOR.C apporte la démonstration d'une boucle de type *for*.

Listing 8 : Le programme FOR.C

```
[1 ]  /* Chapitre 3.3 */
[2 ]
[3 ]  /* Nom du programme: FOR.C    */
[4 ]  /* Notre première boucle FOR */
[5 ]
[6 ]  main()
[7 ]  {
[8 ]      int index;
[9 ]
[10]      for(index = 0;index < 6;index = index + 1)
[11]      { /* Début de la boucle FOR */
[12]         printf("La valeur de ");
[13]         printf("index est %d\n",index);
[14]      } /* Fin de FOR */
[15]  }
```

Ce type de construction permet de programmer des boucles de manière très compacte. Cette technique est particulièrement adaptée au C.

Venons-en au programme proprement dit. La boucle *for* s'étend sur les lignes 10 à 14, et est introduite par le mot clé *for*, suivi d'une expression écrite entre parenthèses. Des points virgule décomposent cette expression en trois champs. Le premier, qui contient l'expression

```
index = 0
```

est le champ d'initialisation. C'est ici qu'une valeur initiale est affectée à la variable d'incrémentation : dans notre exemple, elle est mise à 0. La variable d'incrémentation impose la condition initiale requise pour entrer dans la boucle. Dans ce champ, chaque affectation de valeur s'effectue avant (!) le premier cycle de boucle. Plusieurs initialisations peuvent être mentionnées dans ce champ, à condition d'être séparées par des virgules. Aucune limite n'est imposée au nombre des initialisations. Il n'en demeure pas moins que le contenu de ce champ devrait être le plus simple possible.

Le deuxième champ renferme le test de condition d'entrée dans la boucle. Dans notre cas, il consiste à vérifier si la valeur de la variable *index* est inférieure à 6.

Cette condition sera vérifiée avant chaque parcours de la boucle. Dans ce champ, vous pouvez mentionner tout expression dont il est possible de vérifier la valeur de vérité ("vrai" ou faux"). Dans le chapitre suivant, nous entrerons plus dans le détail des valeurs "vrai" et "faux". Toujours est-il que nous avons à faire ici à des boucles testées en entrée.

L'expression contenue dans le troisième champ sera exécutée après (!) un cycle de boucle, c'est-à-dire que les instructions contenues dans la boucle sont exécutées avant elle. De même que dans le premier champ, vous pouvez y mentionner plusieurs expressions séparées par des virgules.

A la suite de l'instruction *for()* peuvent être mentionnée toute instruction ou série d'instructions qui composeront le corps de la boucle.

Une suite d'instructions, également appelée instruction composée (en anglais : compound statement) est un groupe d'instructions C écrit entre accolades. Une série d'instructions est traitée de la même façon qu'une instruction unique.

Dans notre exemple, il s'agit de deux instructions du genre *printf* qui figurent dans les lignes 12 et 13.

Peut-être vous interrogerez-vous sur l'utilité de deux types de constructions dont les effets sont les mêmes. En fait, chacune des constructions de boucles présente ses avantages pour des applications données. C'est ainsi que la boucle *while* sera systématiquement utilisée lorsque vous ne savez pas si une boucle doit être parcourue et, dans l'affirmative combien de cycles devront être effectués. La boucle *for* présente de plus l'avantage d'une structure de contrôle qui peut être programmée de manière très compacte (initialisation, condition d'exécution et modification des variables d'incrémentation). C'est à vous qu'il appartient de décider de la construction la mieux adaptée au problème que vous tentez de résoudre.

Lancez le programme.

Voici l'affichage retourné par le programme FOR.C :

```
La valeur de index est 0
La valeur de index est 1
La valeur de index est 2
La valeur de index est 3
La valeur de index est 4
La valeur de index est 5
```

Comme vous pouvez le constater, les trois programmes de démonstration précédents résolvent en fait le même problème. Dans chacun des cas, une boucle est parcourue six fois, et à chaque cycle la valeur de la variable d'incrémentation est renvoyée à l'écran. Le grand avantage de la boucle *for* réside dans la compacité de sa structure. C'est pourquoi nous avons fréquemment utilisé cette dernière dans nos programmes C.

3.4. Structure de renvoi - L'instruction if

Le programme qui suit servira d'exemple pour notre premier renvoi conditionnel en C.

Listing 9 : Le programme IFELSE.C

```
[1 ] /* Chapitre 3.4 */
[2 ]
[3 ] /* Nom du programme : IFELSE.C */
[4 ] /* Exemple d'une instruction IF et d'une instruction IF..ELSE */
[5 ]
[6 ] main()
[7 ] {
[8 ]   int valeur;
[9 ]
[10]   for(valeur = 0;valeur < 10;valeur = valeur + 1)
[11]   { /* Début de  boucle FOR */
[12]     if (valeur == 2)
[13]         printf("VALEUR est à présent égal à %d\n",valeur);
[14]
[15]     if (valeur < 5)       /* si oui */
[16]         printf("VALEUR est à présent égal à %d, donc inférieur à
                 5\n",valeur);
[17]     else                  /* sinon */
[18]         printf("VALEUR est à présent égal à %d, donc supérieur à
                 4\n",valeur);
[19]   } /* Fin de la boucle FOR */
[20] } /* Fin de MAIN */
```

Notez que, dans la ligne 10, nous avons d'abord à faire à une boucle *for* contenant une instruction combinée. Dans cette série, vous pouvez voir deux instructions *if*. Par la même occasion, ce programme est en fait aussi notre premier exemple d'imbrication de structures de contrôle : des renvois conditionnels ont été intégrés dans une structure de boucle.

Le premier renvoi conditionnel se trouve dans la ligne 12. Il est introduit par le mot clé *if* suivi d'une expression écrite entre parenthèses. Ce n'est que si cette

expression est vrai que l'instruction suivante sera exécutée. Si elle est fausse, l'instruction ne sera pas prise en compte, et par conséquent non exécutée.

La syntaxe générale d'un renvoi est donc :

```
si (condition)
    instructions
```

Dans ce cas également, l'instruction unique peut être remplacée par une série d'instructions écrite entre parenthèses. L'expression

```
valeur == 2
```

vérifie si la valeur de la variable est 2. Il y a donc comparaison. Ne confondez surtout pas cette notation avec l'expression

```
valeur = 2
```

Celle-ci attribue la valeur 2 à la variable *valeur*. Dans le chapitre suivant, nous reviendrons en détail sur cette expression.

Venons-en maintenant à l'instruction *if..else* qui s'étend sur les lignes 15 à 18.

A une différence près, la deuxième instruction *if* est identique à la première. Nous utilisons ici un nouveau mot clé (*else*) suivi d'une instruction *printf*. Une telle construction signifie que la première instruction située derrière *if* sera exécutée lorsque l'expression écrite dans les parenthèses *if* est vraie. Dans le cas contraire, c'est l'instruction qui figure derrière *else* qui sera exécutée, et celle placée à la suite de *if* sera ignorée. Par conséquent, dans une construction de type *if..else* il y a toujours une seule instruction qui est exécutée.

Voici la syntaxe générale d'une construction *if..else* :

```
Si (condition)
    Instruction
Sinon
    Autre instruction
```

Lancez le programme.

L'affichage que vous retourne le programme IFELSE.C est le suivant :

```
VALEUR est à présent égal à 0, donc inférieur à 5
VALEUR est à présent égal à 1, donc inférieur à 5
VALEUR est à présent égal à 2
VALEUR est à présent égal à 2, donc inférieur à 5
VALEUR est à présent égal à 3, donc inférieur à 5
VALEUR est à présent égal à 4, donc inférieur à 5
VALEUR est à présent égal à 5, donc supérieur à 4
VALEUR est à présent égal à 6, donc supérieur à 4
VALEUR est à présent égal à 7, donc supérieur à 4
VALEUR est à présent égal à 8, donc supérieur à 4
VALEUR est à présent égal à 9, donc supérieur à 4
```

Le texte renvoyé vous montre que les deux constructions faisant appel à *if* ont été évaluées correctement. La première instruction *if* n'est exécutée qu'une seule fois : lorsque "valeur" vaut 2. Par contre, la construction *if..else* retourne une réponse à chaque parcours de la boucle. La variable "valeur" est en effet soit toujours inférieure à 5, et dans ce cas le programme exécute l'instruction placée immédiatement derrière *if*, soit supérieure ou égale à 5. Dans ce dernier cas, c'est l'instruction située après *else* qui est exécutée.

3.5. Interruption de boucle - Break et Continue

Le programme qui suit montre comment des boucles peuvent être interrompues avant terme.

Listing 10 : Le programme ARRET.C

```
[1 ] /* Chapitre 3.5 */
[2 ]
[3 ] /* Nom du programme: ARRET.C */
[4 ] /* Exemple d'interruption de boucle par BREAK
         et de saut de cycle par CONTINUE */
[5 ]
[6 ]  main()
[7 ]  {
[8 ]    int index;
[9 ]
[10]    for(index = 5;index < 11;index = index + 1)
[11]    { /* Début de FOR */
[12]      if (index == 8)
[13]         break; /* arrêter si index = 8 , sinon continuer */
[14]         printf("Dans la boucle, la valeur de index est à présent
                 %d\n",index);
[15]      if (index==7)
[16]         printf("Il y a maintenant interruption par break !!!!\n\n\n");
```

```
[17]    } /* Fin de FOR */
[18]
[19]
[20]    for(index = 5;index < 11;index = index + 1)
[21]    { /* Début de FOR */
[22]       if (index == 8)     /* Si index = 8,continuer avec le cycle */
[23]          continue;        /* suivant de la boucle */
[24]          printf("Dans la deuxième boucle la valeur de index est à
                     présent %d\n", index);
[25]          if (index ==7)
[26]          { /* Début IF */
[27]             printf("Boucle interrompue par continue; ");
[28]             printf("à 9, on redémarre !!!\n\n");
[29]          } /* Fin IF */
[30]    } /* Fin FOR */
[31] } /* Fin MAIN */
```

Dans la première boucle *for* (qui commence en ligne 10), le programme est invité, dans les lignes 12 et 13, à quitter immédiatement la boucle, dès que la variable *index* possède la valeur 8.

Break permet d'interrompre immédiatement une boucle. Les instructions contenues dans la boucle et qui pourraient suivre ne seront plus exécutées.

Break permet donc de programmer très facilement une interruption de boucle, dès lors qu'une condition non désirée apparaît dans cette boucle.

Dans la boucle suivante de notre programme (qui commence en ligne 20), ce n'est pas la boucle qui est interrompue : il y a uniquement une interruption dans le corps de boucle lorsque la variable *index* possède la valeur 8 et que par conséquent la comparaison

```
index == 8
```

est vraie. Ce résultat est atteint dans l'instruction

```
if (index == 8)
      continue;
```

à l'aide du mot clé *continue*. L'exécution du programme se poursuit par un cycle supplémentaire de boucle. Les instructions dans les lignes 24 à 28 ne sont alors pas exécutées, et seul *index* est incrémenté de 1, puis l'exécution se poursuit avec la première instruction contenue dans le corps de boucle.

Le mot clé *continue* interrompt par conséquent un cycle de boucle, mais pas la boucle elle-même.

Voici l'affichage retourné par le programme ARRET.C :

```
Dans la boucle, la valeur de index est à présent 5
Dans la boucle, la valeur de index est à présent 6
Dans la boucle, la valeur de index est à présent 7
Il y a maintenant interruption par break !!!!!!!

Dans la deuxième boucle la valeur de index est à présent 5
Dans la deuxième boucle la valeur de index est à présent 6
Dans la deuxième boucle la valeur de index est à présent 7
Boucle interrompue par continue; à 9, on redémarre !!!

Dans la deuxième boucle la valeur de index est à présent 9
Dans la deuxième boucle la valeur de index est à présent 10
```

Comme vous pouvez le constater, la première boucle est réellement arrêtée. La seconde boucle n'est arrêtée que lorsque *index* est égal à 8. Dès que *index* reprend les valeurs 9 et 10, les instructions de la boucle sont à nouveau exécutées.

3.6. Différenciations de cas à l'aide de switch

Dans le programme qui suit, vous découvrirez une nouvelle commande, très performante, et qui présente de multiples facettes : la commande *switch*. Cette structure de contrôle permet de programmer des différenciations de cas.

Listing 11 : Le programme SWITCH.C

```
[1 ] /* Chapitre 3.6 */
[2 ] /* Nom du programme : SWITCH.C */
[3 ] /* Programme de démonstration du mode de fonctionnement
       de l'instruction SWITCH */
[4 ]
[5 ]
[6 ] main()
[7 ] {
[8 ]   int index;
[9 ]
[10]   for (index = 3;index < 13;index = index + 1)
[11]
[12]   { /* Début boucle FOR */
[13]
[14]     switch (index)
[15]
[16]     { /* Début SWITCH */
[17]
[18]       case 3 :printf("La valeur de index est trois\n");
[19]               break;
```

```
[20]      case 4 :printf("La valeur de index est quatre\n");
[21]             break;
[22]      case 5 :
[23]      case 6 :
[24]      case 7 :
[25]      case 8 :printf("La valeur de index est comprise entre 5 et 8 \n");
[26]             break;
[27]      case 11:printf("A présent, sa valeur est 11 \n");
[28]             break;
[29]      default:printf("Les valeurs comprises entre 8 et 11, ou
                        supérieures à 11, ne sont plus définies\n");
[30]      break;
[31]     } /* Fin de SWITCH   */
[32]
[33]    } /*  Fin de la boucle FOR */
[34]  } /*  Fin main()        */
```

Dans notre programme de démonstration, la valeur de la variable *index* est fixée à 3 dans une boucle de type *for*, puis incrémentée de 1 à chaque cycle de cette boucle. La répétition a lieu aussi longtemps que *index* est inférieur à 13. Par conséquent, pendant l'exécution du programme, cette variable prendra toutes les valeurs comprises entre 3 et 12 (extrémités incluses).

Dans le corps de boucle, une différenciation de cas est obtenue à l'aide de la structure *switch*. Cette structure commence en ligne 14 par le mot clé *switch* suivi d'une variable écrite entre parenthèses dont la valeur actuelle sert à différencier des cas. Il s'en suit que l'expression

```
switch(index)
```

signifie : "vérifie le contenu de la variable *index*".

Une accolade ouvrante, qui signale l'ouverture de la structure, est suivie d'instructions qui sont toutes introduites par le mot clé *case*. Après case, vous voyez une valeur suivie d'un double point. Le mot clé *case* introduit chaque différenciation de cas en fonction de la variable à contrôler. Ainsi,

```
case 3:
```

signifie : "si la valeur de la variable à contrôler (*index*, dans notre cas) est égale à 3, exécute toutes les instructions qui suivent le double point. Si par contre la valeur de *index* n'est pas égale à 3, alors ignore toutes les instructions qui suivent le double point."

Par conséquent, dans notre exemple, si la variable *index* prend la valeur 3 à l'occasion d'un cycle de la boucle, l'instruction *printf* renverra à l'écran la ligne :

```
La valeur de index est trois
```

L'importance du mot clé *break* est primordiale. Cette instruction fait en sorte que toutes les instructions qui suivent dans la structure *switch* seront ignorées. Voyons cela plus concrètement sur l'affichage renvoyé par notre programme.

Lancez le programme.

L'affichage retourné par le programme SWITCH.C est :

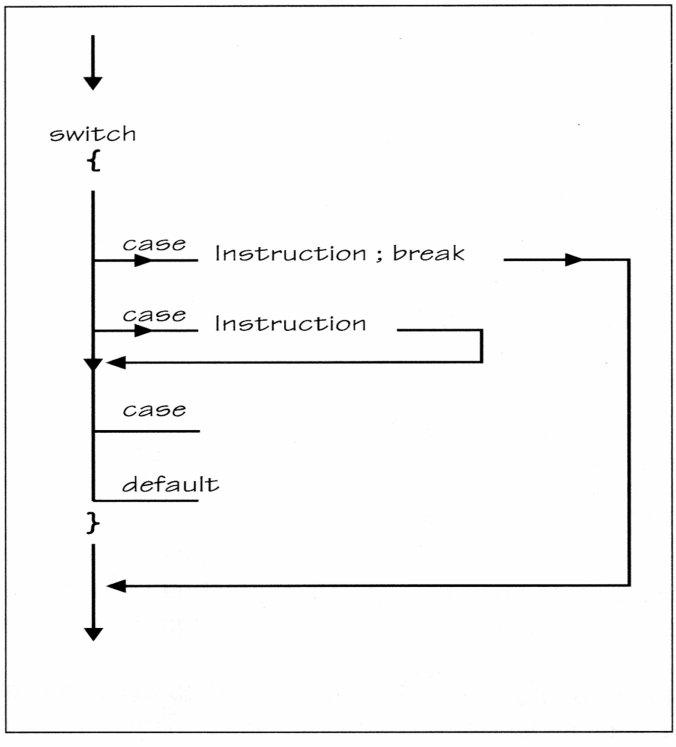
```
La valeur de index est trois
La valeur de index est quatre
La valeur de index est comprise entre 5 et 8
La valeur de index est comprise entre 5 et 8
La valeur de index est comprise entre 5 et 8
La valeur de index est comprise entre 5 et 8
Les valeurs comprises entre 8 et 11, ou supérieures à 11,
ne sont plus définies
Les valeurs comprises entre 8 et 11, ou supérieures à 11,
ne sont plus définies
A présent, sa valeur est 11
Les valeurs comprises entre 8 et 11, ou supérieures à 11,
ne sont plus définies
```

Comme vous pouvez le constater, lorsque *index* prend les valeurs 3 et 4, le programme renvoie les lignes 18 et 20 prévues à cet effet. Est ensuite retourné quatre fois le texte mentionné dans la ligne 25, prévu pour le cas où *index* prend la valeur 8. La cause de cette répétition provient du fait que, pour les valeurs comprises entre 5 et 7, aucune instruction *break* n'est intégrée au programme. C'est pourquoi, lorsque *index* prend l'une des valeurs 5, 6 ou 7, l'instruction de la ligne 25 est systématiquement exécutée. Ce n'est que dans la ligne 26 qui suit, que la structure *switch* est quittée au moyen de *break*.

Mais alors, comment se fait-il que, dans l'affichage qui suit, la ligne

```
Les valeurs comprises entre 8 et 11, ou supérieures à 11,
ne sont plus définies
```

soit renvoyée deux fois ? Cet affichage est programmé dans la ligne 29, à l'aide du mot clé *default* appartenant à la structure *switch*. Celui-ci fait en sorte que, pour toutes les valeurs que peut prendre *index*, et qui n'ont pas été introduites par *case* (c'est-à-dire les valeurs 9, 10 et 12), l'instruction programmée derrière *default* est exécutée.



La structure switch

Les constructions de type *switch* peuvent, elles aussi, être imbriquées à volonté. *Switch* est une commande très puissante, qu'il convient de bien assimiler, afin de l'utiliser intensivement dans vos propres programmes.

3.7. Renvoi inconditionnel à l'aide de goto

Le programme qui suit présente l'utilisation du renvoi inconditionnel à l'aide de *goto*. L'usage en est simple : chaque mot clé *goto* est suivi du nom de l'emplacement du programme auquel vous désirez vous rendre. Ce nom fait office de repère de renvoi (ou label), et peut être placé à un endroit quelconque dans le source du programme. Une marque de renvoi est caractérisée par le double point qui la suit.

Il convient de respecter les deux règles suivantes lorsque l'on utilise *goto* :

Tout renvoi dans une boucle est interdit (vous ne pouvez pas "sauter" en plein milieu d'une boucle). Par contre, vous pouvez quitter une boucle par un renvoi.

Il n'est pas possible de "sauter" d'une fonction dans une autre à l'aide de GOTO.

Listing 12 : Le programme EX_GOTO.C

```
[1 ] /* Chapitre 3.7 */
[2 ] /* Nom du programme : EX_GOTO.C */
[3 ]  /* Exemple de saut non conditionnel avec GOTO  */
[4 ]
[5 ] main()
[6 ] {
[7 ]   int chien,chat,souris;
[8 ]
[9 ]   goto start;          /* va au point de départ */
[10]   quelque_part:        /* c'est ici que se trouve le repère de renvoi
                               quelque_part */
[11]
[12]   printf ("Ceci est une autre ligne dans le chaos:\n");
[13]   goto fin;            /* à présent rends-toi à la fin */
[14]
[15]
[16]   /* ceci est la partie du programme qui contient une commande
          GOTO convenable */
[17]
[18]   start:               /* c'est ici que se trouve le point de départ */
[19]
[20]   for(chien = 1;chien < 6;chien = chien + 1)
[21]   { /* Début la boucle extérieure FOR, "Boucle_chien"  */
[22]
[23]     for(chat = 1;chat < 6;chat = chat + 1)
[24]     { /* Début de la première boucle imbriquée */
[25]
[26]       for(souris = 1;souris < 4;souris = souris + 1)
[27]       { /* Début de la boucle imbriquée centrale */
[28]
[29]         printf("Chien = %d  Chat = %d Souris = %d\n",chien,
                 chat ,souris);
[30]         if ((chien + chat + souris) > 8 ) goto suffit;
[31]       }; /* Fin Boucle_souris */
[32]     };   /* Fin Boucle_chat   */
[33]   };     /* Fin Boucle_chien  */
[34]
[35]   suffit:
[36]   printf("Ainsi s'achève la ballade dans le zoo domestique.\n");
[37]   /* ceci est la fin de la partie du programme qui contient une
          commande GOTO convenable */
[38]
[39]   printf("\nCeci est la première ligne de code spaghetti.\n");
```

```
[40]    goto vers_ici;
[41]
[42]    vers_ou:
[43]    printf("Ceci est la troisième ligne du code spaghetti.\n");
[44]    goto quelque_part;
[45]
[46]    vers_ici:
[47]    printf("Ceci est la deuxième ligne du code spaghetti.\n");
[48]    goto vers_ou;
[49]
[50]    fin:
[51]    printf("En voilà assez, maintenant.\n");
[52]
[53] } /* Fin de main */
```

Le programme précédent est un exemple typique de ce que l'on appelle "code spaghetti", et montre pourquoi de nombreux auteurs de logiciels tentent au maximum d'éviter les instructions *goto* dans leurs programmes. La seule utilisation judicieuse de *goto* dans ce programme se trouve en ligne 30, dans laquelle une commande unique permet de sortir d'une boucle à triple imbrication.

L'analyse de notre exemple de programme est difficile, en raison des instructions *goto* programmées dans les lignes 9, 13, 30, 40, 44 et 48 qui nous promènent vers l'avant et l'arrière dans le programme. Dans la ligne 9, nous sommes renvoyés à l'étiquette *start:* de la ligne 18, c'est-à-dire au début d'une boucle *for* à triple imbrication. Ces boucles ne sont toutefois pas exécutées complètement. La ligne 30

```
if ( (chien + chat + souris) > 8 ) goto suffit;
```

renvoie avant terme au label *suffit:* de la ligne 35. Cette étiquette est située en-dehors des boucles, dont l'exécution est donc arrêtée. Dans la ligne 40, nous sommes renvoyés à la ligne 46, à l'étiquette *vers_ici*, puis dans la ligne 48, à l'étiquette *vers_ou* de la ligne 42. Ensuite, l'instruction *goto* de la ligne 44 nous envoie à *quelque_part* dans la ligne 10, puis, en passant par la ligne 13, vers *fin:* dans la ligne 50. Dans cette section, le programme s'achève alors après une dernière instruction d'affichage.

Voici l'affichage retourné par le programme EX_GOTO.C

```
Chien = 1  Chat = 1  Souris = 1
Chien = 1  Chat = 1  Souris = 2
Chien = 1  Chat = 1  Souris = 3
Chien = 1  Chat = 2  Souris = 1
Chien = 1  Chat = 2  Souris = 2
Chien = 1  Chat = 2  Souris = 3
Chien = 1  Chat = 3  Souris = 1
```

```
Chien = 1  Chat = 3  Souris = 2
Chien = 1  Chat = 3  Souris = 3
Chien = 1  Chat = 4  Souris = 1
Chien = 1  Chat = 4  Souris = 2
Chien = 1  Chat = 4  Souris = 3
Chien = 1  Chat = 5  Souris = 1
Chien = 1  Chat = 5  Souris = 2
Chien = 1  Chat = 5  Souris = 3
Ainsi s'achève la ballade dans le zoo domestique.

Ceci est la première ligne de code spaghetti.
Ceci est la deuxième ligne du code spaghetti.
Ceci est la troisième ligne du code spaghetti.
Ceci est une autre ligne dans le chaos.
En voilà assez, maintenant.
```

3.8. Structures de contrôle liées

Bien que limité en performances, le programme CELSIUS.C qui suit présente une mise en relation des structures que, jusque-là, nous avons étudiées individuellement. Ce programme génère un tableau de correspondance de valeurs des échelles thermométriques Celsius et Fahrenheit, et affiche les points de solidification et d'évaporation de l'eau (l'idée de ce programme est extraite du livre "The C Programming Language" de Kernighan et Ritchie).

Listing 13 : Le programme CELSIUS.C

```
[1 ] /* Chapitre 3.8 */
[2 ]
[3 ] /* Nom du programme :CELSIUS.C */
[4 ]
[5 ]
[6 ] /*****************************************************************/
[7 ] /*                                                               */
[8 ] /* Ceci est un programme de conversion de températures           */
[9 ] /* écrit en langage C. Ce programme génère et affiche un         */
[10] /* tableau d'équivalences entre les échelles de températures     */
[11] /* Fahrenheit et Celsius, ainsi que les températures             */
[12] /* de solidification et d'évaporation de l'eau.                  */
[13] /*                                                               */
[14] /*****************************************************************/
[15]
[16] main()
[17] {
[18]   int index;          /* Variable de suivi de boucle */
[19]   int fahrenheit;     /* Température en degrés Fahrenheit */
[20]   int celsius;        /* Température en degrés Celsius */
[21]
[22]   printf("\t\tTable de correspondanceCelsius/Fahrenheit\n");
```

```
[23]   printf("\t\t[redacted]\n")
[24]
[25]
[26]   for(index = -2;index <= 12;index = index + 1)
[27]   { /* Début de boucle  */
[28]      celsius = 10 * index;
[29]      fahrenheit = 32 + (celsius * 9)/5;
[30]      printf("\t\t  C =%4d   F =%4d ",celsius,fahrenheit);
[31]      if (celsius == 0)
[32]          printf("\t\tPoint de solidification de l'eau");
[33]
[34]      if (celsius == 100)
[35]          printf("\t\tPoint d'évaporation de l'eau");
[36]          printf("\n"); /* Saut de ligne après chaque ligne */
[37]   } /* Fin de boucle */
[38] } /* Fin de MAIN */
```

L'organisation du code du programme est, ici, particulièrement importante. La structure est l'image de la logique du programme, et facilite la recherche d'erreurs. L'expérience aidant, vous développerez votre propre style de programmation. L'exemple précédent devrait être une incitation au développement de votre style personnel.

Au début du listing, vous trouvez une description rapide du programme. Dans le corps du programme, vous constatez que les instructions *for* et *if* sont mises en évidence par indentation du texte. Le retrait à gauche des instructions *printf* contenues dans les lignes 32 et 35 signale que cette instruction appartient à une instruction autre que la boucle *for*, située à l'intérieur de celle-ci : l'instruction *if*.

Dans ce programme, nous faisons appel à trois variables : *index* pour contrôler la boucle, *fahrenheit* pour les températures de l'échelle Fahrenheit, et *celsius* pour les températures de l'échelle Celsius. Chacune de ces variables est déclarée comme variable entière.

La nouveauté qui apparaît dans ce programme réside dans les opérations mathématiques des lignes 28 et 29. Nous utilisons à cet effet ce que l'on appelle des opérateurs. Il s'agit là de symboles qui permettent de relier des variables. Suivant l'opérateur utilisé, la relation ainsi établie fournit un certain résultat. Les éléments que l'on relie par des opérateurs sont appelés opérandes. Ainsi, dans l'instruction

```
celsius = 10 * index
```

le symbole *"*"* est un opérateur. La constante 10 et la variable *index* sont des opérandes. Le résultat de la combinaison des opérandes et de l'opérateur est affecté à la variable *celsius* par l'intermédiaire de

l'opérateur d'affectation (le double signe d'égalité). Etant donné que le symbole "*" est utilisé pour la multiplication, cette ligne signifie que la variable *celsius* est le résultat de la multiplication de la valeur de *index* par la constante 10 ;

Le symbole "+" désigne l'addition, et le symbole "/" représente la division. Par conséquent, l'instruction de la ligne 29

```
fahrenheit = 32 + (celsius * 9) / 5
```

signifie que la valeur attribuée à *fahrenheit* pour la première valeur *celsius* (qui est -20) est :

```
32 + (-20 * 9) / 5 = 32 +(-180 / 5) = 32 +(- 36) = -4
```

Pour toutes les valeurs de l'échelle Celsius comprises entre 20 degrés et 120 degrés, ce calcul permet d'obtenir le tableau suivant.

Voici l'affichage retourné par le programme CELSIUS.C :

```
C = -20        F =  -4
C = -10        F =  14
C =  0         F =  32        Point de solidification de l'eau
C =  10        F =  50
C =  20        F =  68
C =  30        F =  86
C =  40        F = 104
C =  50        F = 122
C =  60        F = 140
C =  70        F = 158
C =  80        F = 176
C =  90        F = 194
C = 100        F = 212        Point d'évaporation de l'eau
C = 110        F = 230
C = 120        F = 248
```

3.9. Récapitulation

Le langage de programmation C connaît des constructions de boucles à cycle obligatoire et à cycle facultatif. Les boucles à cycle facultatif (testées en entrée) vérifient la condition d'exécution de boucle avant (!) d'entrer dans la boucle. Les boucles à cycle obligatoire (testées en sortie) ne vérifient cette condition qu'après un premier cycle de boucle. Les boucles à cycle facultatif, également appelées boucles testées en entrée, peuvent être programmées à l'aide des mots clé *while* et *for*. L'avantage des boucles programmées avec *while* réside dans le fait qu'il n'est pas nécessaire de savoir si la boucle doit être parcourue et, dans l'affirmative, combien de fois le cycle doit être répété. Une construction utilisant *for* permet une programmation très compacte, par concentration des variables d'incrémentation et de contrôle à un même endroit. Les boucles testées en sortie, et qui sont parcourues au moins une fois, sont programmées à l'aide des mots clé *do..while*. Lorsque le corps d'une boucle comporte plus d'une instruction, il doit être délimité par des accolades. Le mot clé *break* interrompt immédiatement l'exécution d'une boucle. Le mot clé *continue*, par contre, n'interrompt qu'un cycle de boucle.

Les renvois sont programmés à l'aide des mots clé *if..else*. Les diverses branches de la structure peuvent, ici aussi, contenir plus d'une instruction, et celles-ci devront aussi être écrites entre accolades. Un renvoi inconditionnel est programmé à l'aide du mot clé *goto*. Dans la mesure du possible, il est préférable d'éviter les renvois inconditionnels dans les programmes.

Les différenciations de cas sont programmées à l'aide des mots clé *switch..case..break..default*. Le mot clé *switch* doit systématiquement être suivi de la variable à tester, écrite entre parenthèses. *Case* introduit les valeurs à distinguer. Chaque série d'instructions à exécuter pour une valeur donnée doit être conclue par *break*. *Default* sera toujours suivi des instructions qui seront systématiquement exécutées lorsqu'aucune des valeurs définies par *case* ne se présente.

Dans ce chapitre, nous avons abordé la programmation de toutes les structures de contrôle disponibles en C. Dans les chapitres qui suivront, nous allons découvrir tout ce qui concerne les opérateurs et les structures de données utilisables en C.

Exercices se rapportant au chapitre 3

Exercice 1 :
Ecrivez un programme qui renvoie cinq fois votre nom à l'écran. Vous écrirez ce programme en trois exemplaires, en utilisant à chaque fois une autre structure de boucle.

Exercice 2 :
Ecrivez un programme qui compte de 1 à 10, et qui renvoie chaque nombre du décompte dans une nouvelle ligne. Vous intégrerez à ce programme des instructions qui, pour les valeurs 3 et 7, affichent les lignes suivantes

```
Voici le nombre 3
Voici le nombre 7
```

Exercice 3 :
Modifiez le programme SWITCH.C de manière à ce qu'il ne renvoie que les valeurs paires de la variable *index*.

4 Affectations de valeurs et comparaisons logiques

Dans ce chapitre, nous parlerons de variables et de l'affectation de valeurs à des variables. A cette occasion, vous découvrirez tous les types de données implémentés en C. De plus, vous apprendrez à afficher les valeurs de variables à l'écran suivant certains formats. La programmation de comparaisons logiques et le traitement des valeurs de vérité constituent un deuxième point important de ce chapitre. Nous accéderons ainsi progressivement aux fondements de la conception de programmes C plus volumineux.

4.1. Affectation de valeurs entières

Le programme qui suit nous servira d'exemple pour la déclaration de variables entières et les diverses techniques d'affectation de valeurs.

Listing 14 : Le programme SIMPLE.C

```
[1 ] /* Chapitre 4.1 */
[2 ]
[3 ] /* Nom du programme : SIMPLE.C */
[4 ] /* Exemple d'affectation de valeurs entières - */
[5 ] /* Ce programme ne renvoie rien à l'écran */
[6 ]
[7 ] main()
[8 ] {
[9 ]   int a,b,c;    /* Variables entières a, b et c */
[10]
[11]   a = 12;
[12]   b = 3;
```

```
[13]   c = a + b;    /* Addition simple */
[14]   c = a - b;    /* Soustraction simple */
[15]   c = a * b;    /* Multiplication simple */
[16]   c = a / b;    /* Division simple */
[17]   c = a % b;    /* Calcul du reste d'une division */
[18]
[19]   c = 12*a + b/2 - a*b*2/(a*c + b*2); /* Expressions mathématiques */
[20]   c = c/4+13*(a + b)/3 - a*b + 2*a*a; /* complexes */
[21]
[22]   a = a + 1;          /* Incrémentation d'une variable */
[23]   b = b * 5;
[24]
[25]   a = b = c = 20;     /* Affectation multiple */
[26]   a = b = c = 12*13/b;
[27]   a = a * c /10;
[28] }
```

Dans la ligne 9 de ce programme trois variables sont déclarées comme variables entières. Les lignes 11 et 12 présentent des affectations simples de valeurs aux variables *a* et *b*. Dans les quatre lignes qui suivent, vous voyez la traduction en C des fonctions arithmétiques élémentaires. Dans la ligne 17, vous découvrez l'affectation par l'intermédiaire de l'opérateur *Modulo* (%). Cet opérateur retourne le reste d'une division dont les opérandes sont *a* et *b*. L'usage de cet opérateur est restreint aux données entières. Dans les deux lignes qui suivent, vous voyez comment l'on peut combiner des opérandes (ici, les variables *a*, *b* et *c*) et des opérateurs (fonctions arithmétiques élémentaires) en des expressions mathématiques complexes.

Les lignes 22 et 23 présentent des affectations de lignes correctes qui incrémentent une variable d'une constante, par addition ou par multiplication. Dans la section qui suivra, nous montrerons que de telles affectations peuvent être écrites de façon plus compacte.

Dans les lignes 25 et 26, qui dans l'immédiat pourraient vous sembler inhabituelles, se trouvent des constructions qui exploitent la possibilité du compilateur consistant à savoir lire une commande de la droite vers la gauche.

Que se passe-t-il dans la ligne 25 ?

Le compilateur se met à lire de la droite vers la gauche, trouve la valeur 20, et affecte cette valeur à la variable *c* par l'intermédiaire de l'opérateur d'affectation "=". Le compilateur poursuit sa lecture et rencontre

```
b = c
```

l'instruction suivant laquelle il doit affecter à la variable *b* le résultat de la dernière opération. Ce résultat est la valeur contenue dans la variable *c*, à laquelle vient d'être attribuée la valeur 20. A présent, la variable *b* contient donc, elle aussi, la valeur 20. En poursuivant sa lecture, le compilateur affecte enfin la valeur de *b* à la variable *a*. En fait, dans une seule ligne, une valeur a pu ainsi être affectée à trois variables distinctes.

La ligne 26 montre qu'il est non seulement possible d'affecter plusieurs fois une constante à plusieurs variables, mais également le résultat d'une opération complexe. Il est d'autre part possible d'utiliser des variables dans l'expression figurant à l'extrême droite d'une affectation multiple, et même les variables auxquelles des valeurs sont attribuées (soit, dans notre cas, les variables *a*, *b* et *c* !).

Notre programme de démonstration ne renvoie rien à l'écran. Toutefois, étant donné que vous avez appris comment renvoyer des valeurs entières à l'écran en C, vous devriez essayer de développer ce programme à l'aide de la fonction *printf()*, de manière à ce que les résultats produits par les diverses instructions apparaissent sur votre écran.

4.2. Autres types de données

Le programme qui suit montre l'utilisation de nouveaux types de données en C. La découverte de ces types de données étant pour l'instant notre soucis essentiel, ce programme ne contient pas non plus d'instruction de renvoi vers l'écran. Les lignes de commentaires précisent à chaque fois les changements de type des variables.

Listing 15 : Le programme TYPESUPP.C

```
[1 ] /* Chapitre 4.2 */
[2 ]
[3 ] /* Nom du programme : TYPESUPP.C */
[4 ] /* Autres déclarations de types en C  */
[5 ]
[6 ] main()
[7 ] {
[8 ]   int a,b,c;              /* de -32768 à 32767 sans partie décimale */
[9 ]   char x,y,z;             /* -128 bis 127 sans partie décimale */
[10]   float un,deux,trois;    /* de 3.4E-38 à 3.4E+38 avec partie décimale */
[11]
[12]   a = b = c = -27;
[13]   x = y = z = 'A';        /* A est une constante alphanumérique,
                                  pas le nom d'une variable */
```

```
[14]   un = deux = trois = 3.6792;
[15]
[16]   a = y;          /* a vaut à présent 65 (Code ASCII de la lettre A) */
[17]   x = b;          /* x vaut à présent -27 */
[18]   un = b;         /* un vaut à présent -27.00 */
[19]   a = deux;       /* a vaut à présent 3 */
[20]
[21] }
```

Dans ce listing, vous relevez la présence du type de données *int* que vous connaissez déjà, mais également les types de données *char* et *float*, auxquels nous allons plus particulièrement nous intéresser.

Les données de type *char* sont des données dont la valeur est entière. Etant donné qu'un unique octet est réservé pour ce type de données en mémoire, sa représentation est comprise entre -128 et 127. Les données de type *char* sont généralement utilisées pour représenter des caractères ASCII ou des lettres.

Lorsqu'à une variable doit être affectée une lettre, celle-ci devra être écrite entre apostrophes. Cette technique permet de signaler que 'A' est une lettre, et non le nom d'une variable. 'A' est une constante dite littérale ou monocaractère. Lorsque plusieurs caractères sont écrits entre apostrophes, l'on parle d'une constante alphanumérique.

Dans notre programme, les données de type *float* constituent un nouveau type de données : les nombres à virgule flottante. Il s'agit là de valeurs numériques appartenant à un intervalle de grande amplitude, et pouvant comporter un grand nombre de décimales significatives. Les variables de type *float* sont mémorisées dans plusieurs octets (variable d'un ordinateur à l'autre). En Quick C et Turbo C, c'est-à-dire sur des compilateurs développés pour PC, la plage des valeurs s'étend de 3.4E-38 à 3.4E+38 (3,4 * 10^38). Il s'agit là de nombres pouvant comporter jusqu'à 37 chiffres. La précision du calcul va jusqu'à sept chiffres significatifs derrière le séparateur décimal.

Dans les lignes 12 à 14, des valeurs sont attribuées aux variables précédemment définies, valeurs compatibles avec le type de données. Par la suite, de nouvelles affectations modifient les valeurs de ces variables. A cette occasion, il y a même échange de valeurs entre des variables de types différents.

4.3. Conversion de variables

Consacrons quelques lignes aux possibilités de conversion des données de types *char* et *int*. Etant donné que les variables de type *char* constituent en fait un sous-ensemble de l'ensemble des variables entières, la plupart des fonctions C prévues pour traiter des variables entières fonctionnent aussi avec des données de type *char*. Lorsque l'on active une telle fonction avec une variable de type *char*, le programme convertit simplement cette variable en une variable entière, et poursuit ensuite normalement son travail.

C'est la raison pour laquelle vous pourrez pratiquement mélanger à votre guise ces types de données, sans qu'aucune erreur ne se produise. Le compilateur ne sera pas pris au dépourvu, dans ce cas (Mais vous, peut-être...). Il est donc préférable de ne pas trop faire appel à cette possibilité. Utilisez plutôt le type de données qui convient, et vous ne serez jamais confronté à aucune difficulté.

Etant donné que les données de type *char* sont en fait des variables entières, rien ne s'oppose à l'attribution de la valeur d'une variable de type *char* à une variable entière, et c'est d'ailleurs ce qui se produit dans la ligne 16. Dans l'autre sens, il convient d'être plus prudent ! Lorsque vous affectez le contenu d'une variable de type *int* à une variable de type *char*, vous risquez de créer un "dépotoir de données" lorsque la valeur de la variable entière n'appartient pas à l'intervalle compris entre -128 et +127. Aucun problème n'est toutefois à craindre si la variable entière appartient à cet intervalle. La ligne 18 montre qu'il est extrêmement simple d'affecter le contenu d'une variable entière à une variable de type *float*. Pour la réciproque, c'est une autre histoire : étant donné que les variables *float* peuvent comporter une partie décimale, il faut, dans le cas d'une affectation à une variable entière, préciser ce qu'il doit advenir des chiffres situés derrière la virgule. Implicitement, le langage C "coupe" simplement la partie décimale.

Ce programme ne renvoie rien à l'écran, et, pour l'instant, vous ne connaissez pas encore les formes sous lesquelles les données de types *char* et *float* peuvent être visualisées. Dans l'immédiat, nous vous prions d'être patient, et de vous contentez de comprendre les lignes qui composent le programme.

4.4. Types de données supplémentaires

Voici un programme qui présente encore d'autres types de données, et plus particulièrement, qui explique comment ces divers types de données peuvent être renvoyés à l'écran. Vous verrez dans ce listing tous les types de données standard disponibles en C. Il existe toutefois aussi des types de données plus complexes (tableaux - en anglais : arrays -, et structures) auxquels nous nous consacrerons en détail dans les chapitres 8 et 13.

Listing 16 : Le programme MULTITYP.C

```
[1 ] /* Chapitre 4.3 */
[2 ]
[3 ] /* Nom du programme : MULTITYP.C */
[4 ] /* Déclaration de tous les types de données standard en C  */
[5 ]
[6 ] main()
[7 ] {
[8 ]  int a;              /* Variable entière simple          */
[9 ]  long int b;         /* Variable entière de type Long    */
[10]  short int c;        /* Variable entière de type Short   */
[11]  unsigned int d;     /* Variable entière sans signe relatif */
[12]  char e;             /* Variable de type Char            */
[13]  float f;            /* Variable à virgule flottante     */
[14]  double g;           /* dito, avec double précision      */
[15]
[16]
[17]  a = 1023;
[18]  b = 2222;
[19]  c = 123;
[20]  d = 1234;
[21]  e = 'X';
[22]  f = 3.14159;
[23]  g = 3.1415926535898;
[24]
[25]  printf("a = %d\n",a);  /* Affichage décimal          */
[26]  printf("a = %o\n",a);  /* Affichage octal            */
[27]  printf("a = %x\n",a);  /* Affichage hexadécimal      */
[28]  printf("b = %ld\n",b); /* Affichage décimal long     */
[29]  printf("c = %d\n",c);  /* Affichage décimal court    */
[30]  printf("d = %u\n",d);  /* Affichage sans signe relatif*/
[31]  printf("e = %c\n",e);  /* Affichage d'un caractère   */
[32]  printf("f = %f\n",f);  /* Affichage d'un nombre à virgule flottante */
[33]  printf("g = %f\n",g);  /* Affichage double précision  */
[34]  printf("\n");          /* Saut de ligne              */
[35]  printf("a = %d\n",a);  /* Affichage INT simple       */
[36]  printf("a = %7d\n",a); /* Largeur de champ de 7 caractères */
[37]  printf("a = %-7d\n",a);/* alignement à gauche sur 7 caractères */
[38]  printf("\n");          /* Saut de ligne */
[39]  printf("f = %f\n",f);  /* Affichage "float" simple */
```

```
[40]  printf("f = %12f\n",f);        /* Largeur de champ de 12 caractères */
[41]  printf("f = %12.3f\n",f);      /* 3 décimales */
[42]  printf("f = %12.5f\n",f);      /* 5 décimales */
[43]  printf("f = %-12.5f\n",f);     /* Alignement à gauche */
[44] }
```

La première variable déclarée est de type *int*. Vient ensuite une variable de type *long int*, nombre entier pour lequel quatre octets sont réservés en mémoire de travail, et qui peut par conséquent prendre une valeur comprise entre -2.147.483.648 et 2.147.483.647. La variable qui suit, de type *short int* prend ses valeurs dans le même intervalle que les données de type *int*, c'est-à-dire entre -32.768 et +32.767. Les données de type unsigned *int* occupent, elles aussi, deux octets en mémoire de travail, et disposent du même intervalle de définition que *int*, mais ne possèdent toutefois pas de signe relatif. Leur plus grande valeur est par conséquent 65.535 et la plus petite est 0. Les variables de types *char* et *float* nous sont déjà connues. Il ne nous reste donc plus qu'à voir les données de type *double*. Les données de type double sont des nombres à virgule flottante s'étendant sur un large intervalle compris entre 1.7E-308 et 1.7E+308.

Revenons au programme. A la suite des déclarations des types de données, des valeurs sont affectées aux variables afin de pouvoir apporter la démonstration des possibilités d'affichage dans la suite du programme.

A partir de la ligne 25, vous voyez une série de descripteurs de champs, tels qu'on les utilise dans l'instruction*printf*. En voici la liste :

Code de conversion	Signification
d	Affichage en notation décimale
o	Affichage en notation octale
x	Affichage en notation hexadécimale
u	Affichage sans signe relatif (type unsigned)
c	Affichage sous forme de type char
s	Affichage sous forme de chaîne de caractères (string)
f	Affichage sous forme de nombre à virgule flottante (float)
p	Affichage sous forme d'adresse mémoire (pointeur)

Ces caractères sont précédés du symbole % qui précise qu'il s'agit là de codes de conversion. Entre le signe % et le code de conversion, vous pouvez encore faire figurer les caractères suivants, dont la signification est particulière :

Code de conversion	Signification
- (Tiret)	Aligner à gauche dans le champ
. (Point)	Emplacement du séparateur décimal
l	Affichage en notation décimale longue

Chacun de ces codes de conversion est utilisé dans ce programme en guise d'exemple. Les codes de conversion s (pour le renvoi des chaînes alphanumériques) et p (pour le renvoi d'un pointeur dirigé sur une variable) font toutefois exception : nous y reviendrons en détail dans la suite.

A présent, vous êtes en mesure de renvoyer à l'écran les types de données que vous connaissez dans le format de votre choix. Essayez maintenant de modifier le programme de démonstration MULTITYP.C de manière à ce que le contenu de chaque variable soit renvoyé dans une ligne.

Dans la section qui suit, nous allons voir comment programmer des comparaisons logiques en C, et plus particulièrement la manière dont laquelle le langage C interprète une valeur comme étant vraie ou fausse.

4.5. Comparaisons logiques

Dans le programme qui suit, vous trouverez de nombreux exemples de programmation de comparaisons logiques en C. La logique est extrêmement importante pour résoudre efficacement des problèmes, et la méconnaissance de l'utilisation et des effets produits par les opérateurs logiques interdit toute programmation de qualité. C'est pourquoi nous vous conseillons d'étudier de manière approfondie le programme de démonstration qui suit.

Listing 17 : Le programme LOGIQUE.C

```
[1 ] /* Chapitre 4.5 */
[2 ]
[3 ] /* Nom du programme: LOGIQUE.C */
[4 ] /* Programme de démonstration de comparaisons logiques */
[5 ]
[6 ] main()
```

```
[7 ] {
[8 ]   int x = 11,y = 11,z = 11;
[9 ]   char a = 40,b = 40,c = 40;
[10]   float r = 12.987,s = 12.987,t = 12.987;
[11]
[12]   /* Premier groupe de comparaisons logiques */
[13]
[14]   if (x == y) z = -13;       /* Attribue la valeur -13 à z */
[15]   if (x > z)  a = 'A';       /* Attribue la valeur 65 à a  */
[16]   if (!(x > z)) a = 'B';     /* Pas de modification        */
[17]   if (b <= c) r = 0.0;       /* Attribue la valeur 0.0 à r */
[18]   if (r != s) t = c/2;       /* Attribue la valeur 20 à t  */
[19]
[20]   /* Deuxième groupe de comparaisons logiques */
[21]
[22]   if (x = (r != s)) z = 1000; /* x est un nombre positif    */
[23]                               /* et z reçoit la valeur 1000 */
[24]   if (x = y) z = 222;         /* x reçoit la valeur de y    */
                                   /* et 222 est attribué à z    */
[25]   if (x != 0) z = 333;        /* 333 est attribué à z       */
[26]   if (x) z = 444;             /* 444 est attribué à z       */
[27]
[28]   /* Troisième groupe de comparaisons logiques   */
[29]
[30]   x = y = z = 77;
[31]   if ((x == y) && (x == 77)) z = 33;   /* z reçoit la valeur 33 */
[32]   if ((x > y) || (z > 12))   z = 22;   /* z reçoit la valeur 22 */
[33]   if (x && y && z) z = 11;             /* z reçoit la valeur 11 */
[34]
[35]   if ((x = 1) && (y = 2) && (z = 3)) r = 12.00;
[36]     /* x reçoit la valeur 1, y reçoit la valeur 2,       */
[37]     /* z reçoit la valeur 3 et r reçoit la valeur 12.00 */
[38]
[39]   if ((x == 2) && (y = 3) && (z = 4)) r = 14.56;
[40]     /* Rien ne change */
[41]
[42]   /* Quatrième groupe de comparaisons */
[43]
[44]   if (x == x); z = 27.345;  /* z change toujours      */
[45]   if (x != x)  z = 27.345;  /* Rien ne change         */
[46]   if (x = 0)   z = 27.345;  /* x reçoit la valeur 0, z ne change pas */
[47] }
```

Le programme commence par la déclaration de 9 variables auxquelles il est immédiatement attribuées des valeurs (cela est nouveau, pour vous). Une telle procédure, qui consiste à regrouper dans une même instruction la déclaration et l'affectation d'une valeur, est appelée "initialisation".

Dans les lignes 14 à 18 vient ensuite le premier groupe de comparaisons. Celui-ci présente la technique de comparaison la plus simple, puisqu'on se contente d'y confronter deux valeurs de variables. Dans la ligne 14, notez la différence entre le signe d'égalité simple et double. L'expression

```
if(x == y)
```

vérifie si la valeur de *x* est égale à celle de *y*. Un signe d'égalité unique, comme dans

```
z = -13
```

attribue à la variable située à gauche du signe d'égalité la valeur placée à sa droite. Par conséquent, la signification de la ligne 14 est la suivante :

Lorsque la valeur de la variable *x* est la même que celle de la variable *y*, alors affecte la valeur -13 à la variable *z*.

Dans les lignes 15 à 18, vous trouvez les opérateurs de supériorité (>), de négation (! , = "non"), d'infériorité au sens large (<=, "inférieur ou égal"), ainsi que l'opérateur "différent de" (!=). A chaque fois que la valeur de la comparaison est vraie (dans les paragraphes qui suivent vous apprendrez comment cette valeur de vérité est déterminée et exprimée), l'instruction qui suit immédiatement la comparaison est exécutée. Voici les significations des lignes qui contiennent de telles instructions :

Ligne 15 :
Lorsque la valeur de la variable *x* est strictement supérieure à celle de la variable *z*, alors attribue la constante littérale '*A*' à la variable *a*.

Ligne 16 :
Lorsque le contenu de la variable *x* n'est pas strictement supérieur à celui de la variable *z*, alors attribue à la variable *a* la constante littérale '*B*'.

Ligne 17 :
Lorsque le contenu de la variable *b* est inférieur ou égal au contenu de la variable *c*, alors attribue la valeur 0 à la variable *r*.

Ligne 18 :
Lorsque le contenu de la variable *r* est différent de celui de la variable *s*, alors divise le contenu de la variable *c* par la constante 2, puis attribue le résultat obtenu à la variable *t*.

4.5.1. Valeurs de vérité

Les comparaisons qui constituent le second groupe (de la ligne 22 à la ligne 26) sont déjà plus complexes. Pour pouvoir les comprendre, il nous faut approfondir le sens des termes "vrai" et "faux" dans le langage de programmation C.

En C, le "vrai" logique est un nombre entier quelconque non nul. Le "faux" logique est défini comme étant la valeur 0.

En conséquence, chaque valeur entière ou de type char peut être choisie comme résultat d'une comparaison logique, mais on peut aussi considérer que la valeur de vérité représente une variable interne, sans nom, et créée par le programme. Reprenons cette explication plus concrètement en nous basant sur la ligne 22 du programme.

```
if (x = (r != s)) z = 1000;
```

Observez la première proposition de comparaison. Rappelons qu'il s'agit toujours de l'expression qui suit directement le mot clé *if*. L'expression

```
(r != s)
```

renvoie un nombre non nul, puisque, dans la ligne 17, la valeur 0.0 a été affectée à la variable *r*. Par conséquent, d'après les conventions établies en C, cette expression est vraie. Cette valeur positive implicite (c'est-à-dire choisie par le programme) est ensuite affectée à la variable *x* par

```
(x = (r != s))
```

Etant donné que le résultat de la comparaison *if* est vrai (puisque *x* est un nombre entier positif), la valeur 1000 est attribuée à la variable *z*. L'instruction de la ligne 22 est donc suivie de deux conséquences :

① Une valeur différente de 0 est attribuée à la variable *x*

② La valeur 1000 est attribuée à la variable *z*.

Ce type de raisonnement étant nouveau, nous vous conseillons de bien réétudier cette ligne avant de passer aux explications portant sur la ligne suivante. Pour tous vos raisonnements futurs, retenez tout particulièrement que :

Un nombre entier différent de 0 spécifie qu'une expression ou une variable est vraie. La valeur 0 la signale comme étant fausse.

Ligne 24 :

```
if (x = y) z = 222;
```

Cette ligne doit vous permettre de vérifier si vous avez bien compris les explications précédentes. Dans cette ligne, la valeur positive de *y* (c'est-à-dire 11) est attribuée à la variable *x*. Par conséquent la comparaison

```
if (x=y)
```

est vraie et la valeur 222 sera attribuée à la variable *z*. Les exemples supplémentaires qui composent les lignes 25 et 26 répètent une fois encore le principe que nous venons de décrire.

4.5.2. Liaisons logiques

Dans les lignes 30 à 33, vous découvrez des concepts de comparaison supplémentaires tels que le "*et*" logique (en anglais : *and*) et le "*ou*" logique (en anglais : *or*). Dans un premier temps, les trois variables entières sont initialisées avec la valeur 77. Dans la ligne 31 se trouve une comparaison faisant appel au symbole *&&*, le "*et*" logique. L'instruction devra être lue de la façon suivante :

Si *x* est égal à *y* et si *x* est égal à 77, alors, et seulement dans ce cas, le résultat est vrai. C'est pourquoi la valeur 33 est affectée à la variable *z*.

Dans la ligne suivante, vous découvrez la comparaison logique à l'aide de "*ou*". Le symbole utilisé ici est | |. En fait, ce caractère particulier (symbole de filtre) ne fait pas partie intégrante du clavier français. La technique universelle qui permet de le produire consiste à passer par son code ASCII. A cet effet, maintenez enfoncée la touche <Alt>, et entrez le nombre 124 sur le pavé numérique.

Venons-en à l'instruction proprement dite. Sa signification est :

Si x est supérieur à y ou si z est supérieur à 12, alors attribue la valeur 22 à la variable z.

Etant donné que z est supérieur à 12, le résultat de la première comparaison est sans importance. Voici les conventions dont il convient de tenir compte :

Une liaison réalisée avec l'opérateur logique "*ou*" est toujours vraie lorsque l'une au moins des conditions entrant en jeu est vraie.

Une liaison utilisant l'opérateur logique "*et*" est toujours fausse quand l'une au moins des conditions entrant dans la combinaison est fausse.

4.5.3. Evaluation de comparaisons logiques

L'évaluation d'une condition combinée s'effectue de la gauche vers la droite, jusqu'à ce que le résultat de la comparaison (vrai ou faux) soit établi. Dès que cette certitude est acquise, la vérification est interrompue. Cela signifie que la vérification d'une combinaison de type "*et*" sera interrompue immédiatement lorsque l'une des composantes possède la valeur "faux", puisqu'il suffit d'une proposition fausse pour qu'une combinaison avec "*et*" soit entièrement fausse. Une liaison utilisant "*ou*" sera interrompue dès détection d'une composante dont la valeur de vérité est "vrai", puisqu'il suffit qu'une proposition soit vraie pour qu'une combinaison avec "*ou*" soit vraie.

4.5.4. Priorités d'évaluation

Il s'agit de savoir quel est l'opérateur qui est évalué en premier, et quel est celui que l'on évalue en dernier. Il existe à ce sujet plusieurs règles, mais nous préférons ne pas entrer dans le détail sur ce sujet. Nous vous suggèrons plutôt de choisir vous-même l'ordre des évaluations en plaçant des parenthèses dans vos comparaisons. En effet :

Les parenthèses sont les éléments de priorité du niveau le plus élevé, et sont toujours évaluées en premier lieu.

Vous pouvez vérifier votre degré de compréhension des comparaisons logiques en analysant les lignes 33 à 46 du programme. La ligne 44 mérite toutefois une remarque supplémentaire :

```
if (x == x); z = 27.345;
```

Dans cette instruction, la valeur de *z* changera toujours, puisqu'un point virgule conclut l'instruction *if*. Ainsi, l'instruction qui suit est indépendante de la comparaison qui précède, et sera toujours exécutée. Soyez donc prudent dans l'usage du point virgule !

L'utilisation de comparaisons logiques en association avec l'instruction *if* est extrêmement importante dans la programmation en C, et sa compréhension doit être parfaite si l'on désire rédiger des programmes efficaces et de qualité.

Complétez le programme en y ajoutant les lignes manquantes permettant d'obtenir un affichage à l'écran, et vérifiez par vous-même les effets produits par les opérateurs et les instructions utilisées dans cette application.

4.6. Récapitulation

Le langage C connaît plusieurs catégories de données caractérisées par un type de données. Le type de données impose l'intervalle de définition d'une variable, et dépend du nombre des octets réservé en mémoire à son intention. Le tableau suivant récapitule les types standard de données, avec la capacité qu'ils occupent en mémoire, et leur intervalle de définition.

Type de données	Occupation en mémoire	Domaine de définition
char	1 octet	de -128 à +127
short	int	2 octets
unsigned int	2 octets	de 0 à 65535
long	4 octets	de -2.147.483.648 à +2.147.483.647
float	4 octets	de 3,4E-38 à 3,4E+38
double	8 octets	de 1,7E-308 à 1,7E+308

Les variables sont déclarées par spécification du type de données qu'elles représentent et par indication de leur nom.

L'attribution de valeurs à des variables s'effectue à l'aide de l'opérateur =. Dans une même instruction, une même valeur peut être attribuée à plusieurs variables. Les opérateurs mathématiques fondamentaux sont + pour l'addition, - pour la soustraction, * pour la multiplication, et / pour la division. De plus, l'opérateur % permet de déterminer le reste entier d'une division de valeurs entières.

La valeur affectée à une variable peut être une combinaison mathématique complexe de plusieurs valeurs et/ou de variables. Même la variable à laquelle sera attribué le résultat de la combinaison peut être utilisée dans la formulation. Lors de l'attribution de valeurs le programmeur doit veiller à ce que la valeur attribuée soit conforme au type de données défini pour la variable, il court le risque d'obtenir des résultats imprévisibles.

Les comparaisons logiques renvoient toujours une valeur de vérité. En C, le "vrai" logique est défini comme étant un nombre entier différent de zéro, et le "faux" comme étant le nombre 0. La priorité d'évaluation dans les combinaisons logiques et mathématiques peut être imposée grâce à des parenthèses. Tous les opérateurs disponibles en C sont récapitulés en annexe dans le présent ouvrage.

Exercice se rapportant au chapitre 4

Modifiez le programme LOGIQUE.C de manière à ce que le résultat de l'évaluation logique des variables soit renvoyé à l'écran dans une ligne descriptive.

5 Les opérateurs spéciaux en C

Dans ce chapitre, vous découvrirez des opérateurs du C qui, l'expérience le montre, peuvent poser problème aux néophytes. Ces opérateurs permettent toutefois une programmation efficace et compacte, et caractérisent bien l'esprit des programmes C. De plus, vous découvrirez ici une nouvelle structure de renvoi.

Le langage C dispose d'opérateurs et de structures qui risquent d'être déconcertants pour le néophyte, car il n'est pas possible de les comprendre intuitivement : ils n'ont aucun équivalent dans d'autres langages de programmation. Nous abordons là ce que l'on appelle la partie cryptée du C. Celle-ci contribue pour beaucoup à l'efficacité du code source compilé, et c'est pourquoi elle est utilisée intensivement par les programmeurs C expérimentés. Dans ce chapitre, vous serez en permanence confronté à ces concepts : vous les retrouverez dans de nombreux codes de programmes publiés.

Le programme qui suit vous apporte la démonstration des concepts évoqués ci-dessus.

Listing 18 : Le programme CTYP.C

```
[1 ] /* Chapitre 5 */
[2 ]
[3 ] /* Nom du programme : CTYP.C */
[4 ] /* Exemple d'utilisation d'opérateurs et de constructions
       très usités en C */
[5 ]
[6 ] main()
[7 ] {
```

```
[8 ]   int x = 0, y = 2, z = 1025;
[9 ]   float a = 0.0,b = 3.14159,c = -37.234;
[10]
[11]   /* incrémenter (par addition) */
[12]
[13]   x = x + 1;        /* la valeur de x est incrémentée de 1 */
[14]   x++;              /* incrémenter après accès */
[15]   ++x;              /* incrémenter avant accès */
[16]   z = y++;          /* z = 2, y = 3 */
[17]   z = ++y;          /* z = 4, y = 4 */
[18]
[19]   /* décrémenter (par soustraction) */
[20]   y = y - 1;        /* la valeur de y est décrémentée de 1 */
[21]   y--;              /* décrémenter après accès /*
[22]   --y;              /* décrémenter avant accès */
[23]   y = 3;
[24]   z = y--;          /* z = 3, y = 2 */
[25]   z = --y;          /* z = 1, y = 1 */
[26]
[27]   /* Opérations arithmétiques */
[28]   a = a + 12;       /* 12 est ajouté à a        */
[29]   a += 12;          /* idem                     */
[30]   a *= 3.2;         /* a est multiplié par 3.2 */
[31]   a -= b;           /* b est soustrait à a     */
[32]   a /= 10.0;        /* a est divisé par 10.0   */
[33]
[34]   /* Poser des conditions */
[35]
[36]   a = (b >= 3.0) ? 2.0 : 10.5 ;     /* Cette expression     */
[37]
[38]   if (b >= 3.0)                     /* et celles-ci sont    */
[39]     a = 2.0;                        /* identiques. Les deux */
[40]   else                              /* fournissent le même  */
[41]     a = 10.5;                       /* résultat             */
[42]
[43]     c = (a >) b?a:b;                /* c Maximum de a et b  */
[44]     c = (a >) b?b:a;                /* c Minimum de a et b  */
[45] }
```

Dans un premier temps, des variables entières et à virgule flottante sont déclarées et simultanément initialisées dans les lignes 8 et 9. La première instruction exécutable se trouve en ligne 13. Elle se contente d'ajouter le nombre 1 à la valeur de la variable *x*. Les deux instructions suivantes effectuent la même opération, en faisant toutefois appel aux concepts cryptés évoqués plus haut.

Il s'agit simplement d'une convention spécifique au C qui dit que :

Un double signe d'addition (de soustraction) placé devant ou derrière une variable augmente (réduit) la valeur de cette variable de 1. Le double signe d'addition est appelé opérateur d'incrémentation. Le double signe de soustraction, opérateur de décrémentation.

Autres conventions à connaître :

Si le double signe d'addition (de soustraction) se trouve devant une variable, celle-ci sera augmentée (diminuée) de la valeur 1, c'est-à-dire, en jargon technique, incrémentée (décrémentée), avant (!) d'être utilisée.

Si le double signe d'addition (de soustraction) se trouve derrière la variable, celle-ci sera incrémentée (décrémentée) de 1 après (!) avoir été utilisée.

Dans l'instruction suivante, en ligne 16, la valeur de la variable *y* est affectée à la variable *z*, puis la variable *y* est incrémentée. Dans la dernière instruction de ce bloc, c'est l'inverse qui se produit. La variable *y* est d'abord incrémentée, puis la valeur de *y* est attribuée à la variable *z*, les deux variables contenant dès lors la même valeur.

Le bloc d'instructions qui suit apporte la démonstration du processus inverse. La variable *y* est décrémentée de 1 par l'intermédiaire du double signe de soustraction. Pour le reste, les règles qui s'appliquent sont les mêmes que celles valables pour le double signe d'addition.

5.1. Opérateur arithmétique

Un autre opérateur très utile est l'opérateur arithmétique qui modifie la valeur d'une variable par l'intermédiaire d'une constante quelconque.

Dans la première instruction, que vous connaissez d'ailleurs, la constante 12 est ajoutée à la variable *a* (ligne 28). La même opération se produit dans la ligne suivante, dont la syntaxe est, pour nous, nouvelle. Comme vous pouvez le constater dans les lignes qui suivent, il est possible d'utiliser les quatre opérateurs binaires "+", "-", "/" et "*" sous la forme utilisée ici. L'opérateur arithmétique est placé devant l'opérateur d'affectation "=", lui-même suivi de la constante. Cette technique peut être utilisée avec tous les opérateurs binaires à l'exception de && et | |. L'opérateur et le signe d'égalité ne doivent pas être séparés par un espace. A la droite du signe d'égalité peut se trouver toute expression de syntaxe correcte.

Dans notre programme de démonstration, nous avons volontairement choisi des constantes simples, afin de vous familiariser avec ce nouveau concept. De même que les opérateurs d'incrémentation et de décrémentation, les opérateurs arithmétiques sont, eux aussi, très utilisés par les programmeurs en C.

5.2. Structure if compacte

L'instruction *if* de la ligne 43 constitue une structure typique de la programmation en C. Celle-ci permet une programmation très compacte. La structure se compose de trois éléments. Le double point et le point d'interrogation servent de séparateurs. L'expression située avant le point d'interrogation est considérée comme condition de l'expression globale, et le langage détermine en priorité sa valeur de vérité. Lorsque cette expression est vraie, il y a évaluation de l'expression qui suit le point d'interrogation (branche Oui), ou exécution de l'instruction qui suit ce point d'interrogation. Si la valeur de vérité est "faux", c'est l'expression située derrière le double point (branche Non) qui est évaluée, ou l'instruction qui se trouve derrière ce double point qui est exécutée.

Vous constatez donc qu'il s'agit là d'une version comprimée de la structure de contrôle *if..else*. L'avantage de cette structure réside dans sa compacité, et de la plus grande rapidité d'exécution de programme qui en résulte. Les exemples suivants, qui déterminent respectivement le maximum et le minimum de deux nombres, en apportent la preuve.

Dans la ligne 43 est attribué à la variable *c* le résultat de l'opération

```
( a > b) ? a : b
```

Lorsque la variable *a* est strictement supérieure à *b*, et que la comparaison est donc vraie, le résultat de l'expression globale est celui qui suit le séparateur "?", c'est-à-dire la valeur de la variable *a*. Si par contre l'expression située devant le point d'interrogation est fausse, c'est l'expression qui suit le double point qui sera évaluée. La variable *c* recevrait alors la valeur de la variable *b*. L'utilisation de la structure *if* compacte souffre toutefois d'une restriction : cette structure ne supporte pas les imbrications.

5.3. Récapitulation

Les concepts et opérateurs cryptés sont incompréhensibles intuitivement. Leur grand avantage réside dans la compacité et l'efficacité de la programmation qu'ils permettent d'obtenir.

Les opérateurs ++ et -- augmentent (ou incrémentent) et réduisent (ou décrémentent) effectivement la valeur d'une variable de la valeur constante 1. Lorsque ces opérateurs sont placés devant les variables, celles-ci seront incrémentées/décrémentées avant l'accès aux variables. Lorsque les opérateurs sont placés derrière une variable, celle-ci sera modifiée après avoir été utilisée. Les opérateurs d'incrémentation et de décrémentation sont des opérateurs dits unaires.

L'opérateur arithmétique est une abréviation d'opérateurs binaires (à deux termes). Il modifie la valeur d'un opérande d'une constante qui, elle aussi, peut être le résultat d'une combinaison mathématique quelconque. Tous les opérateurs binaires sont utilisables, à l'exception des opérateurs && et | |.

La structure *if* compacte permet de programmer des relations du type si-alors-sinon, mais ne supporte pas l'imbrication.

Exercices se rapportant au chapitre 5

Exercice 1 :
Ecrivez un programme qui compte de 1 à 10, et qui affiche chaque nombre ainsi que son carré.

Exercice 2 :
Ecrivez un programme qui compte de 1 à 10, et qui renvoie chaque nombre avec son inverse à setp chiffres.

Exercice 3 :
Ecrivez un programme qui compte de 1 à 100, mais qui ne renvoie à l'écran que les valeurs comprises entre 32 et 39.

Exercice 4 :
Modifiez le programme IFELSE.C de manière à utiliser au maximum les constructions et opérandes compacts et spécifiques au C.

Exercice 5 :
Modifiez les programmes du premier exercice du chapitre 3 de manière à utiliser au maximum des opérandes et constructions compacts spécifiques au C.

6 Fonctions

Dans ce chapitre, vous apprendrez tout ce qui concerne la programmation de fonctions en C. Vous y verrez des exemples de programmation classique et moderne de fonctions, de transfert d'arguments de fonctions, de prototypes, et vous découvrirez le travail avec les bibliothèques de fonctions, les domaines de définition de variables, ainsi que la distinction entre les variables locales, globales et de registre. Vous vous initierez également à la technique de programmation récursive et, par la même occasion, au travail avec les débogueurs intégrés aux environnements de développement. Vous acquerrez ainsi des connaissances fondamentales à propos de la gestion interne de la mémoire, et apprendrez le principe de fonctionnement de la pile (Stack).

Chaque programme C renferme l'en-tête main(). Ce programme principal ne fait rien de plus que d'activer des sous-programmes. De tels sous-programmes sont généralement appelés fonctions en C.

Dans ce chapitre, nous décrirons tout ce qu'il convient de savoir à propos de la programmation et de l'utilisation de fonctions. Le programme qui suit est un premier exemple de programme C utilisant des fonctions. Notre programme élève au carré les nombres compris entre 1 et 10, puis détermine la somme de toutes les valeurs calculées.

Listing 19 : Le programme FONCT1.C

```
[1 ] /* Chapitre 6 */
[2 ]
[3 ] /* Nom du programme : FONCT1.C */
[4 ] /* Premier exemple de fonctions en C */
[5 ]
[6 ]
[7 ] int somme;            /* Variable globale somme */
[8 ]
[9 ] main()
[10] {
[11]   int index;
[12]
[13]   debut();            /* Activation de la fonction debut() */
[14]
[15]   for (index = 1;index <= 10 ;index++)
[16]     carre(index);     /* Activation de la fonction carre() */
[17]
[18]     fin();            /* Activation de la fonction fin() */
[19] }
[20] debut()               /* Ceci est la fonction debut */
[21] {
[22]   somme = 0;          /* Initialisation de la variable somme */
[23]   printf("Ceci est l'en-tête du programme\n\n");
[24] }
[25]
[26] carre(nombre)         /* Début de la fonction carre() */
[27] int nombre;
[28] {
[29]   int carre_de_nombre;
[30]
[31]   carre_de_nombre = nombre * nombre;     /* Calcul du carré     */
[32]   somme += carre_de_nombre;              /* addition à "somme" */
[33]
[34]   printf("Le carré de \t %2d est \t%d\n", nombre,carre_de_nombre);
[35] } /* Fin de la fonction carre() */
[36]
[37] fin()     /* Ceci est la fonction Fin */
[38] {
[39]   printf("\nLa somme des carrés est %d\n",somme);
[40] }
```

Dans les programmes précédents, vous avez déjà eu à faire à des fonctions (l'en tête *main()* ou les fonctions *printf()*, par exemple). Pour ce qui concerne *printf()*, l'on parle d'une fonction de bibliothèque (en anglais : library function). Celle-ci est incluse dans le compilateur. Les fonctions que le programmeur définit lui-même sont qualifiées habituellement de fonctions utilisateur, par opposition aux fonctions de bibliothèque.

6.1. Une première fonction - Calcul de sommes et de carrés

A la suite de la déclaration de la variable *somme*, qui, dans la ligne 7, se trouve avant toutes les fonctions, vous voyez que la partie exécutable du programme commence en ligne 13, dans *main()*, par l'instruction *debut()* : une activation de fonctions. Grâce aux parenthèses qui suivent le nom, le compilateur reconnaît qu'il s'agit bien là d'un appel à une fonction, et pas, éventuellement d'une variable mal placée. Dès que, pendant son exécution, le programme rencontre cette ligne, il recherche la fonction qui porte le nom *debut()*, puis exécute les instructions qu'elle contient. Ensuite, le contrôle est transmis à l'instruction qui suit immédiatement l'activation de la fonction, c'est-à-dire celle de la ligne 15.

A cet endroit est programmée une boucle *for* qui est parcourue dix fois de suite, et dans laquelle il est fait appel à une autre fonction : *carre()*. Dans sa conclusion, le programme principal appelle enfin la fonction *fin()*.

Retenons dans un premier temps que le programme est composé de l'en-tête *main()*, d'un appel à la fonction *debut()*, d'une boucle avec dix appels successifs de la fonction *carre()*, et de l'appel à la fonction *fin()*.

Observez d'un peu plus près la fonction *debut()* qui s'étend sur les lignes 20 à 24. Vous voyez le nom de la fonction suivi d'une accolade ouvrante, puis deux instructions C et enfin une accolade fermante. La forme de cette structure nous permet d'établir deux règles concernant les définition de fonctions :

Une fonction est définie par une parenthèse ouvrante et une parenthèse fermante placées à la suite de son nom.

Le corps d'une fonction est délimité par des accolades.

A chaque fois que nous appelons la fonction *debut()*, les deux instructions écrites entre accolades seront exécutées. Cela signifie que la variable *somme* est mise à 0, et, sur l'écran, est renvoyé :

```
Ceci est l'en-tête du programme
```

Revenons au programme principal, et, dans celui-ci, consacrons-nous à la boucle *for*. A l'aide de l'opérateur d'incrémentation ++, nous augmentons la valeur de la variable *index* à chaque cycle de la boucle. L'instruction carre(index) contenue dans cette boucle nous montre une autre caractéristique du C : la transmission

d'une valeur à une fonction (ici, la valeur de *index*). Nous demandons au compilateur de traiter la valeur de *index* dans la fonction *carre()*.

Observons cette fonction qui s'étend sur les lignes 26 à 35. Nous constatons que, dans la ligne 26, une autre variable figure dans les parenthèses de la fonction : la variable *nombre*. C'est sous ce nom qu'est traitée la valeur qui a été attribuée à la fonction lors de son activation. Les variables ou valeurs que l'on transmet à une fonction sont appelées arguments ou paramètres. La fonction *carre()* possède donc un seul argument.

Vous savez qu'en C, il faut toujours faire savoir au compilateur quel type de variable il a à traiter avant de lui demander de travailler sur une variable. C'est la raison pour laquelle le type de la variable *nombre* est déclaré avant la première accolade. En C, ce type de structure est qualifié de style classique. Dans notre prochain programme de démonstration, nous verrons qu'il existe une autre possibilité permettant d'atteindre le même résultat. Retenez par conséquent que :

Lorsque des arguments sont transmis à des fonctions, les types de données de ces arguments doivent d'abord être déclarés.

La fonction "sait" par conséquent que la valeur qui lui sera transmise est un nombre entier. C'est ainsi que la valeur de la variable entière *index* est attribuée à la fonction *carre()*, dans laquelle son traitement sera poursuivi sous le nom *nombre*.

Dans la première instruction de la fonction, qui figure dans la ligne 29, après l'accolade ouvrante, une autre variable entière est déclarée : *carre_de_nombre*. Celle-ci ne peut être utilisée qu'à l'intérieur de la fonction même. En-dehors de la fonction, le programme ne la connaît pas ! Voici donc une quatrième règle qui régit les fonctions :

Les fonctions peuvent posséder leurs propres variables, qui sont déclarées dans une fonction après la première accolade ouvrante.

Dans la fonction, et plus particulièrement dans la ligne 31, la valeur de *carre_de_nombre* est calculée par élévation au carré de la valeur de variable affectée à la fonction. Ensuite, cette valeur est ajoutée à la valeur courante de la variable *somme*. En effet, depuis la section précédente, vous savez que l'instruction :

```
somme+=carre_de_nombre
```

équivaut à l'instruction :

```
somme = somme + Carre_de_nombre
```

Ensuite, les valeurs de *nombre* et de *carre_de_nombre* sont retournées à l'écran, et nous revenons dans le programme principal.

Poursuivons la lecture du programme principal. Nous arrivons maintenant au dernier rappel de fonction : *fin()*. Cette fonction ne possède ni argument, ni variable propre. Elle renvoie simplement une ligne, et utilise à cet effet la valeur de la variable *somme* qui pouvait être utilisée dans toutes les fonctions.

6.2. Arguments de fonctions

Lors de la transmission de l'argument à une fonction, il est extrêmement important de communiquer une copie de la valeur de *index* à la fonction, dont cette dernière pourra poursuivre le traitement dans la variable *nombre*. Dans le cadre de la fonction, cette valeur pourra être modifiée à volonté sans que la variable initiale *index* s'en trouve affectée. Par conséquent, la variable contenue dans le programme principal sera protégée contre tout changement volontaire ou non. Les deux fonctions, celle qui appelle et celle qui est appelée, travaillent par conséquent chacune avec leurs propres variables. Nous verrons dans la suite de ce livre que ce concept est essentiel pour le développement de programmes volumineux. Il est vrai que cette méthode ne nous permet pas de modifier une variable de la fonction appelante. Toutefois, dans le cadre de l'étude des structures de données de type Array (voir chapitre 8) et Pointer (voir chapitre 10), nous apprendrons une technique permettant de réaliser de telles modifications.

Dans la section qui suit, vous verrez comment des fonctions peuvent retourner des valeurs à une fonction appelante. Une valeur ainsi retournée est communément appelée "résultat de la fonction".

6.3. Retourner des résultats de fonctions à l'aide de return

Le programme de démonstration CARRES.C montre comment une fonction renvoie une valeur à la fonction appelante. Le programme calcule le carré des nombres compris entre 0 et 7, et renvoie ces valeurs à l'écran. De plus, vous découvrirez dans ce listing le style moderne de programmation de fonctions, qui a tendance à se généraliser, et que nous utiliserons dans une très large mesure dans la suite de ce livre.

Listing 20 : Le programme CARRES.C

```
[1 ] /* Chapitre 6.3 */
[2 ]
[3 ] /* Nom du programme : CARRES.C */
[4 ] /* Reprise des valeurs d'une fonction - Style moderne de la
       programmation de fonctions */
[5 ]
[6 ]
[7 ] main()    /* C'est ici que commence le programme principal */
[8 ] {
[9 ]   int x,y;
[10]   for(x = 0;x <= 7;x++)
[11]   {
[12]     y = carre(x);  /* y reçoit la valeur déterminée
                           par la fonction carre() */
[13]     printf("Le carré de x = %d est %d\n",x,y);
[14]   }
[15]
[16]   for (x = 0;x <= 7;++x)
[17]       printf("La valeur de x = %d est carré(x) = %d\n",x,carre(x));
[18] }
[19]
[20] carre(int in)          /* Fonction qui élève un nombre au carré */
[21]                        /* la déclaration des variables transmises est */
                            /* faite dans les parenthèses de la fonction   */
[22] {
[23]   int nombre;
[24]
[25]   nombre = in * in;
[26]   return(nombre);      /* carre() retourne la valeur "nombre" */
[27] }
```

Venons-en au programme proprement dit. Dans *main()*, deux variables entières sont déclarées en ligne 9. Ensuite, dans les lignes 11 à 14, vous voyez une boucle *for* qui sera parcourue pour toutes les valeurs allant de 0 à 7. La première instruction de la boucle se trouve dans la ligne 12, et présente un caractère

nouveau. La valeur qui est attribuée à la variable *y* est celle que renvoie la fonction *carre()*. Voyons étape par étape comment cette transmission est réalisée.

Dans un premier temps est transmis comme argument à la fonction *carre()* la valeur de la variable entière *x* dont le traitement sera poursuivi dans la fonction sous le nom *in*. A la différence du style classique, la variable transmise comme argument est déclarée comme variable entière dans les parenthèses de la fonction. Dans la fonction elle-même, vous voyez, en ligne 23, la déclaration de la valeur entière *nombre*, à l'aide de laquelle nous calculons le carré de *in* dans la ligne 25. Le résultat est ensuite renvoyé à la fonction appelante *main()* par l'intermédiaire de l'instruction

```
return(nombre);
```

Dans cette fonction appelante, la valeur ainsi transmise est alors communiquée à la variable *y*. L'instruction

```
y = carre(x);
```

contenue dans la boucle signifie par conséquent :

"attribue à la variable *y* la valeur qui est renvoyée avec l'argument *x* après activation de la fonction *carre()*."

Ce que nous apprend la ligne 26 nous permet donc d'énoncer une cinquième règle :

Des fonctions peuvent retourner une valeur à la partie de programme qui les a activées par l'intermédiaire de l'instruction *return*. Cette valeur est celle déterminée par la fonction. De plus, cette instruction s'accompagne systématiquement de l'interruption de la fonction appelée.

Une autre interprétation possible est celle qui consiste à considérer *carre(x)* comme une nouvelle variable dont le contenu est le carré de la variable *x*. Cette variable peut alors être utilisée partout où pourraient aussi être utilisées les variables que vous connaissez jusque ici. La deuxième boucle *for*, répartie sur les lignes 16 et 17, nous en apporte la preuve : elle se passe totalement de l'appel à une fonction, et utilise comme une variable l'expression *carre(x)* dans la fonction *printf()*. Par rapport à la première boucle, celle-ci est d'une part plus compacte, et d'autre part caractéristique d'un programme C. Nous pouvons par conséquent énoncer une règle supplémentaire :

Les fonctions peuvent être utilisées comme variables. Dans une fonction donnée, il est possible d'activer une autre fonction. Dans ce cas de figure, la fonction "interne" sera traitée en premier lieu, et le résultat qu'elle retourne sera ensuite traité dans la fonction "externe". Il est donc possible d'imbriquer des fonctions.

L'une des conventions que nous avons énoncée précédemment doit toujours être respectée : le type de la variable qui est renvoyée au moyen de *return* (le résultat produit par la fonction) doit être déclaré. Par défaut, le compilateur considère que les résultats renvoyés par les fonctions sont des valeurs entières. Toute autre valeur produite par une fonction doit être déclarée selon son type réel. Dans la section qui suit, nous approfondirons ce sujet.

6.4. Fonctions à virgule flottante

Le programme FLOATQR.C qui suit est un exemple de fonctions dont les valeurs retournées sont de type *float*. Ce programme intègre à la fois le style classique et le style moderne de programmation de fonctions, et calcule les carrés des valeurs comprises entre 0 et 7 sous forme de valeurs à virgule flottante, qui seront ensuite renvoyées à l'écran avec leur partie décimale.

Listing 21 : Le programme FLOATQR.C

```
[1 ] /* Chapitre 6.4 */
[2 ]
[3 ] /* Nom du programme : FLOATQR.C */
[4 ] /* Fonctions avec données de type FLOAT - */
     /* Styles classique et moderne */
[5 ]
[6 ] float z;     /* Ceci est la variable globale z */
[7 ]
[8 ] float sqr(float valent)      /* élève au carré la valeur de type
                                     float transmise, et renvoie un nombre
                                     de type float */
[9 ]                              /* style moderne */
[10] {
[11]   float carre;
[12]
[13]   carre = valent * valent;
[14]   return(carre);
[15] }
[16]
[17]
[18] main()
[19] {
[20]    int index;
```

```
[21]    float x,y,globalsqr();
[22]
[23]    for (index = 0;index <= 7;index++)
        {
[24]        x = index;          /* Convertis int en float    */
[25]        y = sqr(x);         /* y est égal au carré de x */
[26]        printf("Le carré de %d est \t %10.4f\n",index,y);
[27]    }
[28]
[29]    for (index = 0; index <= 7;index++)
[30]    {
[31]        z = index;
[32]        y = globalsqr(); /* y est le carré de z */
[33]        printf("Le carré de %d est \t %10.4f\n",index,y);
[34]    }
[35] }
[36]
[37] float globalsqr()      /* élève au carré et renvoie un nombre
                               de type float */
[38] {                      /* la fonction n'a pas d'argument     */
[39]    return(z*z);        /* elle utilise la variable globale z */
[40] }
```

En début de programme, nous déclarons une variable globale de type *float* dont le nom est *z*. Le qualificatif de "globale" signifie que cette variable est déclarée en-dehors d'une fonction. Elle pourra par conséquent être utilisée dans toutes les fonctions qui suivront. Dans la section suivante, nous nous consacrerons intensivement aux domaines de définition de variables. Pour le présent chapitre, il suffira de savoir que la variable *z* peut être utilisée dans toutes les fonctions.

Suite à cela, dans l'en-tête *main()*, nous déclarons une variable entière et une variable de type *float*. La déclaration de la variable *globalsqr()* est une nouveauté. Il s'agit là d'une activation de fonction pour laquelle nous spécifions qu'une valeur de type *float* sera retournée au programme appelant. En C, c'est là la méthode classique qui permet de définir qu'une fonction doit renvoyer des valeurs qui ne soient pas de type entier.

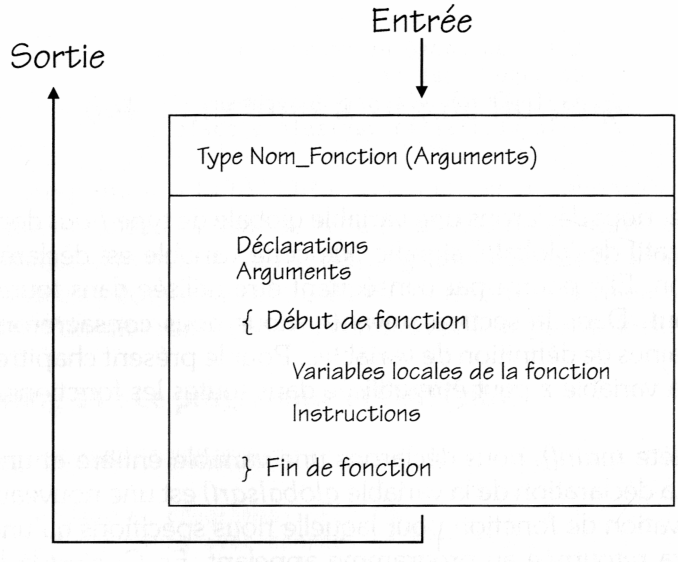
Observons à présent la fonction *sqr()* répartie sur les lignes 8 à 15. Vous voyez que, avant le nom de la fonction, est écrit le mot clé *float*. C'est ainsi que le compilateur "sait" que cette fonction retourna une valeur à virgule flottante. Dans la première instruction de fonction qui suit, en ligne 10, nous déclarons la variable *carre* de type *float*, juste après l'accolade ouvrante qui caractérise le début de la fonction. La ligne

```
carre = valent * valent
```

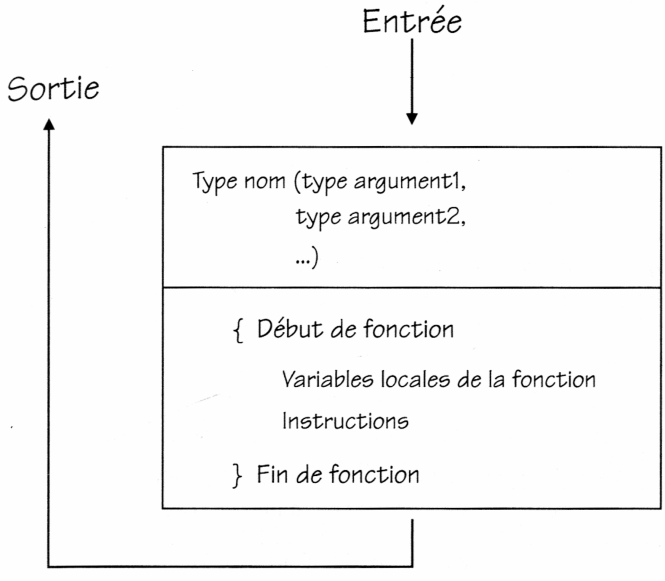
calcule le carré de l'argument *valent* de la fonction, qui est déclaré ici dans les parenthèses de la fonction.

La fonction suivante utilisée dans *main()* s'appelle *globalsqr()*, et est programmée dans les lignes 38 à 41. La valeur de celle-ci est également à virgule flottante, mais, pour le traitement la fonction utilise toutefois la variable globale *z*, ce qui rend sa programmation plus compacte. Etant donné que cette variable est réutilisée directement dans l'instruction *return*, la fonction n'a besoin d'aucune variable propre pour mémoriser transitoirement le produit. Nous pouvons par conséquent nous passer d'une déclaration de variables.

Pour finir, nous pouvons donc retenir deux structures de programmation de fonctions en langage C. Le style "classique" repose sur la structure suivante :



Programmer des fonctions en style "classique"



Programmer des fonctions en style "moderne"

Chaque fonction se caractérise par une "entrée" bien définie, dans laquelle se trouve la déclaration des arguments, ainsi qu'une "sortie" caractérisée par la déclaration de la valeur produite par la fonction.

Dans la section qui suit, nous verrons qu'une séparation stricte des variables déclarées dans une fonction et des variables déclarées en-dehors d'une fonction permet de programmer des fonctions indépendantes les unes des autres.

6.5. Domaines de définition de variables

Il ressort des informations issues des sections précédentes de ce chapitre qu'en C le type d'une variable déclarée est déterminant, puisqu'il délimite son domaine de définition et par conséquent, les possibilités d'utilisation de la variable. Le programme qui suit nous permettra d'approfondir la notion de "domaines de définition de variables". De plus, vous découvrirez dans ce programme l'utilisation de prototypes.

Listing 22 : Le programme DOMAINE.C

```
[1 ] /* Chapitre 6.5 */
[2 ]
[3 ] /* Nom du programme : DOMAINE.C */
[4 ] /* Domaine de définition de variables (globales et locales) - */
     /* Utilisation de prototypes */
[5 ]
[6 ]
[7 ] #include "stdio.h"        /* Prototypes pour les entrées/sorties */
[8 ] void fonction1(void);      /* Prototype pour la fonction1  */
[9 ] void fonction2(void);      /* Prototype pour la fonction2  */
[10] void fonction3(void);      /* Prototype pour la fonction3  */
[11]
[12] int var;                   /* Variable globale var */
[13]
[14] main()
[15] { /* Début de main */
[16]  register int index; /* Variable uniquement disponible dans
                             main(), et mémorisée dans un registre */
[17]
[18]  fonction1();              /* Activation de la fonction1() */
[19]  fonction2();              /* Activation de la fonction2() */
[20]  fonction3();              /* et aussi de la fonction3() */
[21]  /* Boucle "for" de main */
[22]
[23]  for (index = 10;index > 0;index--)
[24]  { /* Début de FOR */
[25]    /* La variable Nombre n'est disponible que dans la boucle interne */
[26]    int nombre;
[28]    for (nombre  = 0;nombre  <= 6;nombre++)
[29]        printf("  %d ",nombre);
[30]        printf("Dans main(), index possède la valeur %d\n",index);
[31]  } /* Fin FOR */
[32] } /* Fin de main */
[33]
[34] int compteur;          /* A partir d'ici, la variable compteur
                               est disponible */
[35] void fonction1(void) /* sans argument ni valeur de fonction */
[36] {
[37]   int index;   /* Variable uniquement disponible dans fonction1 */
[38]
[39]   index = 23;
[40]   printf("La valeur de index dans fonction1() est %d\n",index);
[41] }
[42]
[43] void fonction2(void)
[44] {
[45]   int var;   /* uniquement dans fonction2          */
[46]              /* remplace la variable globale var   */
[47]
[48]   var = 53;
[49]   printf("La valeur de var dans fonction2() est %d\n",var);
[50]   compteur = 77;
[51] }
```

```
[52]
[53] void fonction3(void)
[54] {
[55]   printf("La valeur de compteur dans fonction3() est %d\n",compteur);
[56] }
```

Dans un premier temps, ne tenez pas compte des quatre instructions réparties sur les lignes 7 à 10. Nous y reviendrons plus tard. La première variable déclarée ici (en ligne 12) porte le nom *var* et est une variable globale qui sera disponible dans chaque fonction du programme, puisqu'elle est déclarée avant toutes les fonctions. Une seconde caractéristique de cette variable, réside dans le fait qu'elle reste conservée jusqu'à la fin du programme. Dans la suite du programme, nous déclarons, en ligne 34, la variable *compteur*, également globale, mais non disponible dans l'en-tête *main()*. En effet, elle n'est déclarée qu'après (!) *main()*, mais en-dehors des fonctions programmées après la ligne 34. En conséquence, une variable globale peut être définie comme :

Une variable déclarée en-dehors d'une fonction.

Dans certains cas, cette notion sera également caractérisée de "variable externe".

Revenons à la première déclaration de variables dans *main()*, en ligne 16. Nous déclarons ici la variable entière *index*. Nous verrons la signification du mot clé *register* dans la section suivante. Retenons dans l'immédiat que cette variable n'est disponible que dans *main()*, puisqu'elle a été déclarée à l'intérieur des accolades de la fonction *main()*. Etant donné que cette variable n'est déclarée qu'au moment où *main()* arrive à exécution, et qu'en fin de fonction elle est effacée, on la qualifie aussi de variable automatique. Dans notre cas, *index* n'est effacée qu'en fin de programme car *main()* est actif jusqu'à la fin du programme. En-dehors de *main()*, cette variable est toutefois inconnue dès qu'une fonction appelée est active. Il en va de même avec la variable *nombre* déclarée dans la ligne 26, à l'intérieur de la boucle *for*. Celle-ci est créée en début de boucle, immédiatement après l'accolade ouvrante, et est effacée en fin de boucle. Cela nous permet d'énoncer comme règle :

Chaque partie de programme écrite entre accolades, et par conséquent définie comme entité autonome, peut contenir des définitions de variables qui ne seront valables qu'à l'intérieur de cette partie de programme.

Ces variables ne sont alors disponibles que pour les instructions qui figurent à l'intérieur du bloc de programme, et sont effacées (ou plutôt oubliées) lorsque l'on quitte ce bloc.

6.5.1. Variables automatiques ou locales

Très souvent, l'on qualifie les variables automatiques de variables locales. La fonction *fonction1()* répartie sur les lignes 35 à 41 du programme met en évidence les effets produits et la signification de cette catégorie de variables. Dans cette fonction, nous déclarons une variable *index*, qui n'a rien à voir avec la variable *index* de *main()*. Les deux variables n'ont un sens que dans les blocs de programme dans lesquels elles ont été déclarées. Tant qu'aucune instruction de la fonction *fonction1()* n'est exécutée, c'est-à-dire que cette fonction n'est pas appelée, la variable *index* n'existe pas. Lorsqu'est appelée *fonction1()*, il y a création de la variable *index*, qui sera effacée en fin de fonction. La variable de même nom qui existe dans *main()* n'en est pas affectée. C'est pourquoi les variables locales doivent être réinitialisées à chaque activation de fonctions.

6.5.2. Variables de type registre

Venons-en au mot clé *register*. Un ordinateur peut mémoriser des données dans la mémoire de travail ou dans des registres. Un registre est une unité de mémoire du processeur, et est par conséquent plus rapide que la mémoire de travail. Il est donc judicieux de déclarer comme variables de type registre les variables fréquemment utilisées dans un programme, afin d'en augmenter la vitesse d'exécution. Le nombre des variables de type registre que vous pouvez déclarer dépend de la machine et du compilateur utilisés. Ainsi, le compilateur Turbo C de Borland utilise les registres SI (SI=Source Index) et DI (DI=Destination Index) pour mémoriser des variables déclarées au moyen du mot clé *register*. Il en résulte que, dans un programme donné, seules deux variables peuvent être utilisées comme variables de type registre.

Il ressort de ce qui vient d'être dit que, pour l'exécution d'un programme C, l'emplacement du programme auquel une variable est définie est déterminant :

Les variables qui doivent être de type local pour une fonction ou une partie de programme devront être déclarées après l'accolade ouvrante avant toute instruction exécutable dans cette partie du programme.

6.6. Prototypes

D'un point de vue tout à fait général, un prototype est un exemplaire d'un objet existant réellement. En C, vous avez la possibilité de définir des prototypes de fonctions. Le compilateur utilise ces prototypes pour effectuer une pré-recherche d'erreurs. La vérification porte sur le nombre d'arguments transmis à la fonction lors de son activation, et sur la déclaration de types de ces variables.

Ces précisions étant apportées, nous pouvons revenir aux premières lignes de notre programme. Vous y voyez les prototypes des fonctions définies dans le programme. Le mot clé *void* qui introduit chaque ligne de définition de prototypes fait savoir au compilateur que ces fonctions ne renvoient pas de valeurs. Par conséquent, une instruction du genre

```
index = fonction1();
```

produirait un message d'erreurs. Le mot clé *void* placé dans les parenthèses de la fonction signifie qu'aucun argument ne sera attribué à cette fonction. C'est pourquoi une activation de fonction du genre

```
fonction1(index);
```

renverrait, elle aussi, un message d'erreurs.

Consacrons nous maintenant à l'instruction

```
#include "stdio.h";
```

Cette instruction demande au compilateur d'intégrer le fichier STDIO.H dans le programme courant. Le nom STDIO.H signifie STandarD Input Output, et décrit parfaitement la tâche de ce fichier : mettre à disposition du compilateur des outils tels que des prototypes, des constantes et des macros pour la programmation des entrées/sorties (pour ce qui concerne les macros, nous y reviendrons dans le chapitre suivant). L'extension ".H" est une marque normalisée pour caractériser les fichiers include. Vous connaissez maintenant la raison pour laquelle il fallait définir les chemins d'accès aux sous-répertoires lors de l'installation du compilateur.

6.7. Bibliothèques de fonctions standard

Chaque compilateur dispose d'une quantité plus ou moins grandes de fonctions prédéfinies dont le programmeur peut faire usage. Généralement, il s'agit de fonctions d'entrée/sortie, de fonctions de manipulation de caractères et de chaînes alphanumériques, et de fonctions mathématiques. Les compilateurs adaptés à la création de programmes de qualité professionnelle (tels que Turbo C, C++ et Quick C) proposent d'autre part des fonctions de gestion graphique et du système d'exploitation. De plus, pour toutes ces fonctions, sont définis des prototypes qui peuvent être intégrés dans le code du programme par l'intermédiaire des fichiers Include appropriés.

Les fonctions dont nous venons de parler sont enregistrées dans des bibliothèques (en anglais : libraries) auxquelles le compilateur accède pendant le processus de compilation pour inclure au programme les fonctions qui y sont mentionnées. L'extension des fichiers bibliothèques est ".LIB". Dans le manuel d'utilisation de votre compilateur, vous trouverez une liste des bibliothèques fournie avec le compilateur et des fonctions qu'elles renferment. Grâce à des utilitaires livrés avec le compilateur, vous pouvez ajouter des fonctions aux bibliothèques existantes, ou créer de nouveaux fichiers bibliothèques.

6.8. Récursivité

La récursivité est une technique de programmation qui permet de rédiger des boucles de façon extrêmement compacte. Les programmes qui suivent vous permettront de vous initier à cette technique. Le programme ci-après renvoie à l'écran la série des nombres compris entre 0 et 7, dans l'ordre croissant et décroissant, tout en visualisant le déroulement du programme.

Listing 23 : Le programme RECURSIO.C

```
[1 ] /* Chapitre 6.8 */
[2 ]
[3 ] /* Nom du programme : RECURSIO.C */
[4 ] /* Exemple de programmation récursive */
[5 ] main()
[6 ] {
[7 ]   int index;
[8 ]
[9 ]   index = 8;
[10]   compte_a_rebours(index);  /* Activation de la fonction avec index */
```

```
[11] } /* Fin de main */
[12]
[13] compte_a_rebours(int var)   /* voici la fonction */
[14] {
[15]   var--;
[16]   printf("La valeur de VAR avant arrêt du cycle récursif est
            \t %d\n",var);
[17]   if (var > 0)    /* Condition d'interruption de la récursivité */
[18]     compte_a_rebours(var); /* Auto-activation récursive */
[19]     printf("A présent la valeur de VAR est \t %d\n",var);
                    /* Nous n'en arrivons là que quand la fonction
                       ne s'active plus elle-même./*
[20] } /* Fin de la fonction/*
```

Observons d'abord la fonction *compte_a_rebours()* répartie sur les lignes 13 à 20. Vous voyez, dans la ligne 18, que cette fonction s'appelle elle-même. Cette technique de programmation est appelée récursivité.

Récursivité signifie qu'une fonction s'appelle elle-même.

Il en résulte une boucle pour laquelle il est indispensable de définir une condition d'interruption. Cette condition d'interruption est spécifiée dans les lignes 17 et 18. Ce n'est que lorsque

```
if (var > 0)
```

est vrai que le programme s'appelle lui-même. L'argument transmis à la fonction est le contenu de la variable *var*. Celle-ci est réduite d'une unité à chaque nouvelle activation de la fonction. Etant donné que, dans le programme principal, la valeur initiale de *var* a été fixée à 8 dans la variable *index* (ligne 9), la fonction *compte_a_rebours()* sera activée à la valeur 8 et s'achèvera à la valeur 1.

Afin de pouvoir mieux comprendre les explications qui vont suivre, lancez le programme et analysez les résultats affichés à l'écran.

Voici l'affichage renvoyé par le programme RECURSIO.C :

```
La valeur de VAR avant arrêt du cycle récursif est 7
La valeur de VAR avant arrêt du cycle récursif est 6
La valeur de VAR avant arrêt du cycle récursif est 5
La valeur de VAR avant arrêt du cycle récursif est 4
La valeur de VAR avant arrêt du cycle récursif est 3
La valeur de VAR avant arrêt du cycle récursif est 2
La valeur de VAR avant arrêt du cycle récursif est 1
La valeur de VAR avant arrêt du cycle récursif est 0

A présent la valeur de VAR est 0
A présent la valeur de VAR est 1
A présent la valeur de VAR est 2
```

```
A présent la valeur de VAR est 3
A présent la valeur de VAR est 4
A présent la valeur de VAR est 5
A présent la valeur de VAR est 6
A présent la valeur de VAR est 7
```

Vous constatez que, dans un premier temps, la valeur de *var* est diminuée de 1 (ligne 15), puis avant chaque auto-activation, cette valeur est renvoyée (ligne 16). L'affichage ainsi obtenu est le résultat de la récursivité. Aussi longtemps que *var* est supérieur à 0, la fonction s'appelle elle-même. La dernière instruction de la fonction, celle de la ligne 19, ne peut par conséquent pas être exécutée ! disons plutôt que le programme mémorise cette dernière instruction et la valeur actuelle des variables, (ici, celle de *var*). Ce n'est que lorsque *var* atteint la valeur 0, c'est-à-dire quand la condition d'auto-activation n'est plus remplie, que la fonction ne s'appellera plus elle-même. C'est alors la dernière instruction qui sera exécutée sept fois : une fois pour chaque auto-activation de la fonction. L'affichage a cette fois lieu en sens contraire.

Pour plus de clarté, analysons de façon plus détaillée deux cycles.

1. cycle :	var reçoit la valeur 8 - 1 = 7
Affichage :	la valeur de var avant l'interruption de la récursivité est 7
Var >0	La fonction s'appelle elle-même, et mémorise le dernier résultat renvoyé, soit 7.
2. cycle :	var reçoit la valeur 7 - 1 = 6
Affichage :	la valeur de var avant l'interruption de la récursivité est 6
Var >0	la fonction s'active elle-même et mémorise le dernier résultat renvoyé, car cette fois, la valeur de compteur est 6.
"	
"	
"	
Dernier cycle :	var reçoit la valeur 1 - 1 = 0
Affichage :	la valeur de var avant l'interruption de la récursivité est 0
Var =0	pas d'auto-activation supplémentaire de la fonction. La dernière instruction est répétée plusieurs fois, dans l'ordre chronologique inverse.

Que s'est-il passé ?

A chaque fois que la fonction s'appelle elle-même, elle mémorise toutes les valeurs de variables et les instructions qui complètent la fonction dans une plage de mémoire spéciale : la pile (Stack). A la suite de la dernière activation de la fonction, les instructions ainsi mémorisées sont exécutées en commençant par la suite. Le résultat, vous le voyez à l'écran.

6.9. Gestion de la mémoire sur un exemple de récursivité

Avant l'activation d'une fonction, toutes les données de la fonction appelante sont mémorisées. En jargon informatique, on parle également de sauvegarde des données. Ainsi, après exécution de la fonction appelée, l'état de la fonction appelante peut être restauré. A la fin du sous-programme, l'exécution du programme appelant peut alors se poursuivre là où elle avait été interrompue pour activer le sous-programme.

Ce principe s'applique naturellement aussi aux appels de fonctions récursives. A chaque activation d'une telle fonction, l'ancien état de la fonction est enregistré puis restauré à la fin de l'exécution de la fonction appelée. Cette mémorisation est réalisée selon le principe d'un empilement. Les données sont mémorisées les "unes au-dessus des autres", puis retirées successivement de cette pile (par "lecture"). Ce principe est appelé LIFO : Last In First Out. Les derniers éléments posés sur la pile (on parle plus souvent de Stack) en sont retirés en premier.

C'est ce qui explique pourquoi notre programme de démonstration traite en ordre inverse les données de la dernière instruction de la fonction à la fin de l'exécution de la récursivité.

Pour ce qui concerne la technique de récursivité, il faut toujours veiller à ce qu'une boucle récursive se termine par une condition d'interruption. Il est un autre paramètre à ne pas oublier : la mémoire réservée à la pile est limitée, et il en résulte qu'une récursivité infinie ou des auto-activations de fonctions trop fréquentes peuvent entraîner un débordement de la pile entraînant un plantage du système.

Le programme qui suit montre une application concrète de la technique récursive. Il calcule la factorielle d'un nombre que vous êtes invité à rentrer, puis renvoie cette factorielle à l'écran.

Listing 24 : Le programme FACT.C

```
[1 ] /*********************************************/
[2 ] /*          . R E C U R S I V I T E   en  C    */
[3 ] /*                                             */
[4 ] /*              Calcul de factorielle          */
[5 ] /*                                             */
[6 ] /*********************************************/
[7 ]
[8 ] #include <stdio.h>
[9 ] long i = 1;
[10] int j=0;
[11] long fact(int n);      /* Prototype pour le calcul de factorielles */
[12]
[13] main()
[14] {
[15]    int input;
[16]
[17]    printf("Veuillez entrer un nombre positif \n");
[18]    scanf("%d",&input);/* lecture d'un nombre */
[19]    if (input > 0)
[20]       printf( "\n\nLa factorielle de %d est %lu",input, fact(input));
[21]    else
[22]       printf("Erreur.................................");
[23] }
[24]
[25] long fact(int n)
[26] {
[27]    j++; /* élève la valeur de j */
[28]    if (n == 0)
[29]       return(1);      /* Si n = 0, alors fin de la fonction */
[30]    else               /* sinon */
[31]    {                  /* Début de ELSE */
[32]    printf("La valeur de n dans le cycle %d est %d\n",j,n);
[33]    printf("A été posé sur la pile : \"return
           (%d * fact( %d - 1))\n",n,n);
[34]    return( n * fact (n - 1));  /*  Tant que n>0, poser n sur la pile */
[35]                                /*  et réduire sa valeur de 1         */
[36]    } /* Fin de ELSE */
[37] } /* Fin de la fonction */
```

Ce programme est de type interactif puisqu'après son lancement, l'utilisateur est invité à rentrer un nombre. La programmation de cette saisie se trouve dans les lignes 17 et 18. Dans le chapitre 11, nous décrirons en détail ce qui se produit dans ces lignes.

Dans la ligne 11, vous voyez le prototype d'une fonction *fact()* qui attend une variable entière en guise d'argument, et qui retourne comme résultat une valeur de type *entier long (long integer)*. L'instruction

```
printf("\n\nLa factorielle de %d est %lu",input, fact(input));
```

qui se trouve dans la ligne 20 active la fonction avec la valeur saisie par l'utilisateur dans *input*, puis affiche le contenu de *input* ainsi que le résultat de la fonction activée à l'écran. Une fois de plus, nous avons à faire ici à une imbrication de fonctions typique de la programmation C.

La fonction *fact()*, programmée dans les lignes 25 à 37, calcule à l'aide d'une récurrence la factorielle d'un nombre quelconque transmis à la variable *n*. La structure *if..else* des lignes 28 à 36 fait en sorte que la fonction s'active elle-même aussi longtemps que la valeur de la variable communiquée à la fonction est différente de 0. Pour chaque auto-activation, la ligne

```
return(n * fact(n-1));
```

est posée sur la pile avec les valeurs actuelles de *n*, et, par la même occasion, la fonction est réactivée avec une valeur de *n* réduite de 1. Lancez le programme, et demandez le calcul de la factorielle de 5. L'affichage renvoyé vous permet de suivre l'exécution du programme, et tout particulièrement de voir dans quel ordre les appels de la fonction sont mémorisés dans la pile.

Voici l'affichage retourné par le programme FACT.C lorsque la valeur saisie est 5 :

```
La valeur de n dans le cycle 1 est 5
A été posé sur la pile : "return (5 * fact( 5 - 1))
La valeur de n dans le cycle 2 est 4
A été posé sur la pile : "return (4 * fact( 4 - 1))
La valeur de n dans le cycle 3 est 3
A été posé sur la pile : "return (3 * fact( 3 - 1))
La valeur de n dans le cycle 4 est 2
A été posé sur la pile : "return (2 * fact( 2 - 1))
La valeur de n dans le cycle 5 est 1
A été posé sur la pile : "return (1 * fact( 1 - 1))
La factorielle de 5 est 120
```

Etant donné que, dans la ligne 28, nous avons programmé que lorsque

```
n == 0
```

la fonction doit renvoyer la valeur 1, la pile est traitée de la manière suivante :

① fact(1) est égal à 1 * fact(0), soit 1 *1 = 1

② fact(2) est égal à 2 * fact(1), soit 2 *1 = 2

③ fact(3) est égal à 3 * fact(2), soit 3 *2 = 6

④ fact(4) est égal à 4 * fact(3), soit 4 *6 = 24

⑤ fact(5) est égal à 5 * fact(4), soit 5 *24 = 120

Le résultat renvoyé par le dernier appel de la fonction est donc 120 : la factorielle du nombre 5. Ce résultat apparaît également dans l'affichage programmé à la ligne 20.

Vous avez la possibilité de suivre à l'écran les processus qui se déroulent dans la mémoire de travail grâce aux débogueurs livrés avec les compilateurs. Cette technique vous permet de dégager un premier aperçu des possibilités de recherche et de vérification d'erreurs que permettent ces outils de développement.

6.9.1. Débogage à l'aide du débogueur de Turbo C 2.0 et C++ 3.0

Voyons d'abord ce qu'est un débogueur. Déboguer est synonyme de localiser et corriger les défauts contenus dans un programme (bugs). Un programme qui permet de réaliser cette opération est appelé débogueur (en anglais : Debugger). Jusque ici, nous avons souvent fait du débogage, sans toutefois en citer le nom. Dès que vous demandez la traduction d'un programme et que le compilateur rencontre une erreur de syntaxe, vous faites un débogage : vous localisez une erreur dans le programme et la corrigez.

Toutefois, ce que fait un débogueur est différent du contrôle de syntaxe qu'exerce le compilateur pendant la compilation. Celui-ci recherche les erreurs logiques qui apparaissent lorsqu'un programme fonctionne mais qu'il n'effectue pas les tâches qu'il devrait exécuter. Il existe plusieurs techniques de recherche de telles erreurs. Dans la suite, nous passerons en revue sur des exemples les possibilités mises à notre disposition par le débogueur.

Deux causes sont possibles pour les erreurs logiques de programmation. Il peut d'une part se produire que des variables reçoivent des valeurs qui n'étaient pas prévues. Il est d'autre part possible que certaines parties d'un programme ne soient pas passées en revue pendant l'exécution. Pour pouvoir localiser de telles erreurs, il faut que nous soyons en mesure de suivre les valeurs des variables pendant l'exécution du programme. Parallèlement, il faut que nous connaissions à tout instant la ligne du programme en cours d'exécution. Par la même

occasion, il faudrait aussi que nous puissions arrêter l'exécution du programme. Nous allons simuler toutes ces possibilités à l'exemple du programme FACT.C à l'aide des compilateurs C++ et Turbo C. Le mode d'emploi du débogueur intégré est le même dans ces deux environnements de développement.

Pour que votre test du programme FACT.C corresponde aux descriptions qui vont suivre, il faut que vous entriez la valeur 5 lorsque le programme vous invite à saisir le nombre dont la factorielle doit être calculée.

Plaçons d'abord deux points d'arrêt (Breakpoints). Les points d'arrêts sont des endroits où votre programme devra s'arrêter obligatoirement lors de son exécution. Vous pouvez ainsi l'obliger à stopper à des moments stratégiques, afin de pouvoir visualiser le contenu d'une variable, ou demander l'évaluation d'une expression.

Positionnez le curseur dans la ligne 27 et appuyez sur la combinaison de touches <Ctrl>+<F8>, ou activez l'option *Break/Watch/Toggle Breakpoint* (Turbo C 2.0) ou *Debug/Toggle Breakpoint* (C++). Cette ligne est ainsi repérée comme point d'arrêt. Faites de même dans la ligne 34. Les lignes ainsi repérées sont visualisées en inversion vidéo.

Notre opération suivante consiste à marquer deux variables en leur attribuant la caractéristique Watch. Le contenu des variables ayant reçu la caractéristique Watch est visualisé en permanence dans une fenêtre prévue à cet effet pendant le déroulement du programme. Positionnez dans un premier temps le curseur en ligne 27, sur la variable *j*, et appuyez sur la combinaison de touches <Ctrl>+<F7>, ou activez l'option *Break/Watch/Add Watch* (en Turbo C) ou *Debug/Watch/Add Watch* (en C++). Cette variable possède dorénavant la caractéristique Watch. Répétez la même opération pour la variable *n* de la ligne 34. Si vous travaillez sous Turbo C, une fenêtre s'ouvre automatiquement dans la partie inférieure de l'écran, et vous pouvez y voir les variables qui possèdent la caractéristique Watch. Si tel n'est pas le cas, appuyez sur la touche <F5>. Dans l'environnement de développement de C++, il faut d'abord activer cette fenêtre par l'intermédiaire de l'option *Window/Watch*.

Les messages

```
undefined symbol 'j'
undefined symbol 'n'
```

actuellement affichés dans cette fenêtre signifient que, pour l'instant, les deux variables que nous avons marquées sont inconnues. Les deux variables sont en

effet des variables locales de la fonction *fact()*. Ceci vous prouve que les variables locales ne sont créées par le programme que lorsque la fonction dans laquelle elles sont déclarées est elle aussi active.

Vous pouvez maintenant lancer le programme par pression sur la touche de fonction <F8>, ou par l'intermédiaire de l'option *Run/Step Over*. Le programme sera exécuté jusqu'au point d'arrêt suivant. Répétez plusieurs fois votre appui sur <F8>. Vous voyez qu'après activation de la fonction *fact()* celle-ci s'appelle elle-même dans la ligne 34. Par la même occasion, la valeur des variables *j* et *n* change : j est incrémentée, et n est décrémentée. Dès que n atteint la valeur 0, la comparaison

```
if (n == 0)
```

programmée en ligne 28 est vraie, et la fonction se termine par

```
return(1)
```

Appuyez maintenant sur la combinaison de touches <Ctrl>+<F3>, ou activez l'option *Debug/Call Stack*. Vous voyez la pile, et par voie de conséquence, les fonctions et valeurs qui seront traitées après le dernier appel récursif de la fonction.

```
File   Edit   Run   Compile   Project   Options   Debug   Break/watch
                              Edit
   Line 29    Col 5   Insert Indent Tab Fill Unindent   C:FACT.C
   if (input > 0)                                        Call Stack
      printf( "\n\nLa factorielle de %d est %lu",input, fact  fact(0)
   else                                                    fact(1)
      printf("Erreur ...............................");    fact(2)
}                                                          fact(3)
                                                           fact(4)
long fact(int n)                                           fact(5)
{                                                          main()
   j++; /* élève la valeur de j */
   if (n == 0)
      return(1);     /* Si n = 0, alors fin de la fonction */
   else    /* sinon
   {                    /* Début de ELSE */
   printf("La valeur de n dans le cycle %d est %d\n",j,n);
   printf("A été posé sur la pile : \"return (%d * fact( %d - 1))\n",n,n);
   return( n * fact (n - 1));     /* Tant que n>0, poser n sur la pile
                                     et réduire sa valeur de 1 */
                                 Watch
•n: 0
 j: 6
F1-Help  ↑↓→←-Scroll   <┘-View call                          NUM
```

L'état du programme FACT.C après le dernier appel récursif de la fonction

Actionnez plusieurs fois la touche <F8>. Vous voyez qu'à présent la valeur de n est incrémentée. Lorsque n atteint la valeur 3, appuyez à nouveau sur la combinaison de touches <Ctrl>+<F3>. Vous constatez que la pile a été traitée. Appuyez sur <Esc> pour achever la visualisation de la pile. Si maintenant vous appuyez sur <F7>, ou si vous sélectionnez l'option *Run/Trace Into*, vous pouvez suivre le déroulement du programme, ligne par ligne, en mode Trace. Vous constatez à cet instant que seule la ligne

```
return (n * fact(n-1));
```

est encore traitée. Lorsque la dernière fonction mémorisée dans la pile a été exécutée (c'est-à-dire que *n* possède la valeur 5 qui avait été entrée), l'exécution de la fonction s'achève par renvoi de son résultat à l'écran. Appuyez sur la combinaison de touches <Alt>+<F5>, ou activez l'option *Run/User Screen* (en Turbo C) ou *Window/User Screen* (en C++). Vous voyez l'écran de sortie. Appuyez sur la combinaison de touches <Ctrl>+<F3>. Vous constatez qu'aucune fonction ne se trouve plus dans la pile.

Le débogueur nous a donc permis de suivre le déroulement du programme ligne par ligne ou, sur notre demande, jusqu'à un point d'arrêt. Pendant ce temps, le contenu de deux variables était affiché à l'écran, et avec la combinaison de touches <Ctrl>+<F3>, nous pouvions à tout instant visualiser le contenu de la pile.

A l'occasion d'une nouvelle exécution du programme, nous allons maintenant voir s'il est également possible de changer la valeur d'une variable pendant l'exécution, afin de vérifier les effets produits par cette modification. Exécutez une réinitialisation du programme à l'aide de la combinaison de touches <Ctrl>+<F2> ou par activation de l'option *Run/Program Reset*. "Réinitialiser" signifie remettre à zéro toutes les variables du programme.

Appuyez sur la touche <F8> jusqu'à ce que la valeur de la variable n soit égale à 4, et que le programme s'arrête dans la ligne 27. Actionnez alors la combinaison de touches <Ctrl>+<F4>, ou sélectionnez l'option *Debug/Evaluate* (en Turbo C) ou *Debug/Evaluate/modify* (en C++). Dans les champs respectifs *Evaluate* (Turbo C) ou *Expression* (C++) entrez le nom de variable *n*, et appuyez sur <Return>. Dans le champ *Result*, vous voyez la valeur courante de n, c'est-à-dire 4. Appuyez sur <Curseur bas>, et entrez la valeur 0 dans le champ *New Value*. Validez votre saisie par <Return> puis refermez la fenêtre de saisie par <Esc>.

A présent, lorsque le déroulement du programme se poursuit en mode Trace par pression sur <F7>, la condition

```
n == 0
```

programmée dans la ligne 28 sera vraie, et la récurrence s'achèvera avant terme. Appuyez sur la combinaison de touches <Ctrl>+<F3>. Vous voyez que notre modification de la valeur de *n* a provoqué un changement notable dans l'exécution du programme. Dans la pile, il manque les activations de fact(4), fact(3), fact(2), fact(1) et fact(0). Le résultat définitif fourni par le programme est que la factorielle de 5 est 5. Ce résultat faux provient de l'interruption prématurée de la récursivité dont nous sommes responsables.

6.9.2. Débogage à l'aide du débogueur de Quick C 2.5

Dans cette section, nous allons vous présenter les combinaisons de touches et options de menus qu'il faut utiliser dans le compilateur de l'environnement de développement du Quick C 2.5 de Microsoft pour atteindre les mêmes résultats que précédemment. Nous utiliserons à cet effet le programme FACT2.C. Celui-ci met en évidence le fait que, pour calculer une factorielle, il n'est pas indispensable de faire appel à une récursivité.

Listing 25 : Le programme FACT2.C

```
[1 ] /*****************************************************/
[2 ] /*       Calcul de factorielle sans RECURSIVITE      */
[3 ] /*                                                   */
[4 ] /* calcul de factorielles à l'aide d'une boucle WHILE */
[5 ] /*                                                   */
[6 ] /*****************************************************/
[7 ]
[8 ] #include <stdio.h>
[9 ]
[10] long fact(int n);
[11]
[12] main()
[13] { /* Début de main() */
[14]   int input;
[15]
[16]    printf("Veuillez entrer un nombre positif \n");
[17]    scanf("%d",&input);
[18]    if (input >= 0)
[19]       printf("\n\nLa factorielle de %d est %lu",input,fact(input));
[20]    else
[21]       printf("Erreur ...............................");
```

```
[22]
[23] } /* Fin de main() */
[24]
[25] long fact(int n)
[26] {
[27]   int i = 0;
[28]   long temp = 1;     /* Calcul de résultats intermédiaires */
[29]
[30]   if (n == 0)
[31]       return(1);     /* Si = 0, alors fin de la fonction */
[32]   while (++i <= n)
[33]   {
[34]      temp = temp * i;
[35]      printf("\nDans le cycle %d, la valeur de temp est %d",i, temp);
[36]   }
[37]   return(temp);
[38] }
```

Dans les lignes 32 à 36, vous voyez une boucle *while* qui sera répétée aussi longtemps que la valeur d'une variable *i*, initialisée par 0 dans la ligne 27, sera inférieure ou égale à la valeur d'une variable *n*. Dans la variable *n* est mémorisé le nombre entré par l'utilisateur dont il faudra calculer la factorielle.

Placez à la ligne 34 un point d'arrêt à l'aide de la touche de fonction <F9> ou par activation de l'option *Débogage/Points d'arrêt*. Positionnez ensuite le curseur dans la ligne 34 sur la variable *temp*, et activez l'option *Débogage/Visualiser la variable*. Répétez la même opération pour la variable *i* de la même ligne. Le contenu des deux variables sera dès lors affiché dans la fenêtre de débogage, dans la partie supérieure de l'écran.

Activez ensuite le menu Exécution. Vous voyez ici les diverses options disponibles dans Quick C 2.5 pour poursuivre le pilotage du programme.

```
Fichier Edition Affichage Recherche Make Exécution Débogage Outil Option Aide
                                DEBOG
temp : <Identificateur inconnu>
i : <Identificateur inconnu>
                                C:\TC\FA
                                Initialisation  Maj+F5
                                Exécution          F5
                                Arrêt au curseur   F7
                                Trace pas à pas    F8
                                Trace raccourcie   F10
                                Animation

} /* Fin de main() */

long fact(int n)
{
    int i = 0;
    long temp = 1;     /* Calcul de résultats intermédiaires */

    if (n == 0)
        return(1);     /* Si = 0, alors fin de la fonction */
    while (++i <= n)
    {
        temp = temp * i;
        printf("\nDans le cycle %d, la valeur de temp est %d",i, temp);
    }
    return(temp);
}

F1=Aide  Entrée  Echap=Annuler                          N 00034:023
```

Débogage dans Quick C 2.5

N'oubliez pas, avant de lancer l'exécution de votre programme, de vérifier les chemins des fichiers Include et bibliothèques en activant la commande *Option/Environnement*.

Appuyez plusieurs fois de suite sur la touche <F8>, ou sélectionnez l'option *Trace pas à pas*, ou activez <Alt>+<X>+<P> . A présent, le débogueur traite le programme ligne par ligne. Dès que vous avez entré le nombre 5, vous passez dans la fonction *fact()*, et vous pouvez suivre dans la fenêtre de débogage l'incrémentation de la variable *i*, et l'attribution des factorielles de 1 à 4 à la variable *temp*. A la suite du cinquième cycle de la boucle, *i* possède la valeur 6, la boucle s'arrête, et la valeur de *temp* est retournée à la fonction appelante. A l'aide de la touche <F4> (ou par l'intermédiaire de l'option *Affichage/Ecran de sortie*), nous pouvons maintenant observer le déroulement du programme sur l'écran utilisateur.

Lancez une seconde fois le programme. Entrez une nouvelle fois la valeur 5 pour le calcul de factorielle. Si, pendant l'exécution du programme, vous modifiez le contenu d'une variable, les effets produits sur la suite du déroulement du programme seront visibles. Appuyez sur la touche <F8> jusqu'à ce que les valeurs suivantes soient affichées dans la fenêtre de débogage :

```
temp = 1
i = 2
```

Sélectionnez alors l'option *Débogage/Modifier la variable*. Entrez, pour la variable *i* la valeur 6. Dès que vous ordonnez la poursuite de l'exécution du programme, par pression sur <F8>, vous constatez que la boucle n'est plus exécutée. En effet, puisque la valeur de i est 6, la condition

```
while (i++ <=n)
```

n'est plus remplie. Le programme s'arrête. La conséquence qui en résulte est l'affichage suivant produit par le programme :

```
Dans le cycle 1, la valeur de temp est 1
Dans le cycle 6, la valeur de temp est 6
La factorielle de 5 est 6
```

Le débogueur intégré permet donc de contrôler et de piloter les valeurs attribuées à des variables ainsi que le déroulement du programme. La modification de contenus de variables permet de tester les réactions du programme pour des valeurs données de variables.

Pour finir, voyons comment il est possible de visualiser la pile dans Quick C 2.5. Dans ce but, retirez d'abord le point d'arrêt installé dans la ligne 34 par l'intermédiaire de l'option *Débogage/Points d'arrêt/Purger*. Positionnez le curseur dans la ligne 35, et appuyez sur la touche <F7>, ou activez l'option *Exécution/Arrêt au curseur*. Le programme sera dès lors exécuté jusqu'à la position actuelle du curseur. Après le premier arrêt du programme, sélectionnez l'option *Débogage/Appels*. Vous voyez le contenu de la pile, dans laquelle sont mémorisées les fonctions *fact(5)* et *main()*.

```
Fichier Edition Affichage Recherche Make Exécution Débogage Outil Option Aide
                          DEBOGAGE: fact
temp : 1
i : 1
        printf("                Liste :
                                 fact(5)
} /* Fin de main                 main()
long fact(int n)
{
    int i = 0;
    long temp =
    if (n == 0)
        return(1
    while (++i <                 OK   <Annuler>  < Aide >
    {
        temp = tem
        printf("\nDans le cycle %d, la valeur de temp est %d",i, temp);
    }
    return(temp);
}
F1=Aide  Entrée  Echap=Annuler  Tab=Champ suivant          N 00035:001
```

La pile du programme FACT2.C après activation de la fonction fact()

Les possibilités d'utilisation des débogueurs intégrés aux compilateurs présentés dans ce livre dépassent largement le cadre de ce que nous avons vu dans ce chapitre. Nous espérons toutefois avoir pu vous donner ici les éléments fondamentaux nécessaires à l'utilisation de ces outils extrêmement performants. Au fur et à mesure de vos travaux avec votre compilateur, vous porterez une estime croissante à ces instruments et n'arriverez plus à vous en passer.

6.10. Récapitulation

En C, les sous-programmes sont généralement appelés *Fonctions*. Lors de sa diffusion, un compilateur C est toujours agrémenté de fonctions de bibliothèque prédéfinies, mémorisées dans des fichiers d'extension ".LIB", et pouvant être intégrées dans vos programmes par le compilateur sur votre demande. Le programmeur peut d'autre part écrire ses propres fonctions. Chaque fonction possède un nom, des parenthèses qui lui sont associées, et est toujours délimitée par des accolades.

Lorsque des valeurs sont communiquées à une fonction par la partie de programme appelante (qui peut elle même être une fonction), ces valeurs sont appelées arguments, et leur liste devra être mentionnée dans les accolades de la fonction. Les arguments de fonction doivent être déclarés et séparés par des

virgules. Pour cette déclaration, chaque argument pourra être spécifié individuellement dans les accolades de la fonction (style moderne), ou immédiatement devant la première accolade qui introduit une fonction (style classique). Est toujours transmise en guise d'argument la copie du contenu d'une variable ou une constante, dont le traitement sera ensuite poursuivi dans une variable de la fonction. Par l'intermédiaire de variables globales ou de l'instruction *return()*, des fonctions peuvent renvoyer des valeurs à la partie de programme qui les avait activées. Cette valeur est communément appelée "résultat de la fonction".

Il est possible de définir un prototype pour chaque fonction. Ce prototype regroupe le type de données du résultat de la fonction (implicitement, il s'agit du type *int*), le nom de la fonction et la liste de ses arguments. Le mot clé *void* signale qu'une fonction ne retourne aucune valeur, ou que cette fonction n'attend la transmission d'aucun argument. Les prototypes permettent de réaliser un pré-contrôle de la définition de fonctions, et sont enregistrés dans des fichiers d'en-têtes d'extension ".H" (Header) pour toutes les fonctions de bibliothèques pouvant être intégrées au texte du programme.

Le domaine de définition d'une variable dépend de l'emplacement auquel cette variable est déclarée dans le programme. Les variables qui sont déclarées en-dehors d'une fonction sont appelées variables globales ou externes. Elles peuvent être utilisées dans toutes les fonctions qui suivent leur déclaration. Les variables déclarées à l'intérieur d'un bloc de programme ou d'une fonction sont appelées variables locales ou automatiques. Elles ne sont reconnues qu'à l'intérieur du bloc de programme dans lequel elles ont été déclarées. A chaque activation de fonction, ces variables sont nouvellement créées, puis effacées dès que vous quittez la fonction. Les variables de type registre sont mémorisées dans les registres du processeur. Cette technique permet de les rendre accessibles très rapidement. L'imbrication de fonctions est possible. Dès qu'une fonction s'appelle elle-même, on parle de récursivité.

Les variables générées dans un cycle récursif, ainsi que les instructions non encore traitées sont mémorisées dans une plage spéciale de la mémoire appelée "pile". A la fin de la récursivité, leur traitement est assuré suivant le principe LIFO. LIFO signifie que les dernières informations mémorisées dans la pile (Last In) seront lues d'abord (First Out). Chaque appel doit être conclu par une condition d'arrêt. En l'absence de cette précaution, ou lorsque des quantités trop importantes de données sont mémorisées dans la pile, celles-ci "débordent", et le système se bloque.

Un débogueur permet de piloter et de contrôler l'exécution d'un programme et la valeur des variables qu'il contient. Cette technique permet de localiser des défauts logiques et de les corriger. Les Débogueurs intégrés à Quick C, Turbo C et C++ permettent de placer des points d'arrêt, d'influencer le déroulement d'un programme et de modifier le contenu de variables.

Exercices se rapportant au chapitre 6

Exercice 1 :
Réécrivez le programme CELSIUS.C en programmant une fonction pour déterminer les valeurs de l'échelle Celsius.

Exercice 2 :
Ecrivez un programme qui, à l'aide de la technique de récursivité, renvoie à l'écran les carrés des nombres compris entre 100 et 110.

7 Constantes et macros

Dans ce chapitre, vous apprendrez à définir des constantes et des macros, et à les utiliser judicieusement. Vous verrez que ces éléments contribueront sensiblement à faciliter la lisibilité, et par conséquent la maintenance des programmes.

Les constantes et les macros apportent plus de souplesse à la rédaction des programmes et en facilitent la maîtrise.

Les constantes sont des valeurs qui ne changent pas pendant l'exécution d'un programme, et que le programme ne peut pas modifier, lui non plus.

Imaginons que vous écriviez un programme dans lequel doivent être définies plusieurs valeurs plafond. Dès que les conditions d'exécution de ce programme changent, et qu'il faut une nouvelle valeur plafond, vous êtes contraint de rechercher l'ancienne valeur dans toutes les lignes du programme et de les modifier. Par contre, si vous travaillez avec une constante définie, une modification unique suffit. Voyons concrètement une telle situation grâce au programme DEFINE.C. Ce programme détermine le maximum et le minimum entre deux valeurs, et renvoie le résultat à l'écran.

Listing 26 : Le programme DEFINE.C

```
[1 ] /* Chapitre 7 */
[2 ]
[3 ] /* Nom du programme : DEFINE.C */
[4 ] /* Définition de constantes et de macros */
[5 ]
[6 ]
[7 ] #define START  15     /* Valeur initiale pour la boucle  */
[8 ] #define ENDING 25     /* Valeur finale pour la boucle    */
[9 ]
[10] #define MAX(A,B)  ((A)>(B)?(A):(B)) /* déterminer le maximum */
[11] #define MIN(A,B)  ((A)>(B)?(B):(A)) /* déterminer le minimum */
[12] main()
[13] {
[14]   int index,max,min;
[15]   int count = 20;
[16]
[17]   for (index = START;index <= ENDING;index++)
[18]   {
[19]     max = MAX(index,count); /* Maximum de index, count */
[20]     min = MIN(index,count); /* Minimum de index, count */
[21]     printf("Le maximum est %d, et le minimum est %d\n",max,min);
[22]    }
[23] }
```

Dans ce programme, vous constatez que les lignes 7 à 11 sont toutes introduites par le mot clé *#define*. C'est ainsi que sont définies toutes les constantes et les macros. Considérons la ligne 7 :

```
#define START 15
```

Vous spécifiez ici que la valeur de la constante dont le nom est *START* doit être 15. Suite à cela, le compilateur, avant de traduire le code du programme, recherchera ce nom dans tout le texte et le remplacera par la valeur 15. Dans ce genre de définition, le nom de la constante est donc un descripteur qui améliore la lisibilité du programme.

Cette même remarque vaut également pour la ligne

```
#define ENDING 25
```

Les constantes *START* et *ENDING* sont également appelées constantes symboliques, et l'usage qui s'est répandu consiste à écrire ces constantes en majuscules.

7.1. Définir des macros

Une macro n'est dans l'immédiat rien de plus qu'une définition de constante supplémentaire. La différence avec la constante symbolique réside dans le fait que les macros sont capables d'exécuter des fonctions mathématiques ou des comparaisons logiques.

Voyons à ce sujet la ligne 10 :

```
#define MAX(A,B)  ((A)>(B)?(A):(B))
```

Elle signifie que, dans sa phase de pré-lecture, le compilateur remplace la chaîne de caractères

```
MAX(var1,var2)
```

par l'instruction compacte *if*, en utilisant les variables *var1* et *var2* écrites entre parenthèses en guise d'arguments.

Il s'agit donc là de déterminer le maximum de deux nombres en guise de résultat. Le minimum sera déterminé à l'aide de la seconde macro, présentée dans la ligne 11, dont la construction est semblable à celle de la première.

Vous pourriez évidemment écrire chacune de ces macros sous forme de fonctions. La différence entre les deux techniques réside dans le fait qu'une fonction n'est présentée qu'une seule fois dans le programme compilé, et que chaque activation de fonction s'accompagne nécessairement d'un renvoi. Les macros par contre sont intégrées à tous les emplacements où il y est fait appel. Les renvois deviennent donc sans objet. Le programme est plus long, mais aussi plus rapide.

Etant donné que les macros et les constantes figurent toujours au début d'un programme, les macros et constantes symboliques qui ont fait leurs preuves peuvent être sauvegardées dans des fichiers qui pourront ensuite être intégrés dans d'autres programmes par l'intermédiaire de l'instruction *#include*, sous forme de fichier Include.

Il est essentiel de ne pas conclure les définitions de macros et de constantes par un point virgule. En effet, il ne s'agit pas là d'instructions du programme, mais de propositions implicites destinées au compilateur afin que celui-ci, dans la phase de pré-compilation, remplace les constantes symboliques contenues dans le texte du programme par le contenu correspondant prédéfini par *#define*.

7.2. Ce qu'il ne faudrait pas faire

A prime abord, certaines parenthèses semblent inutiles dans les définitions des macros. Ce n'est toutefois pas le cas. Le programme MACRO.C en apporte la preuve. Celui-ci détermine la surface totale de cubes et de carrés.

Listing 27 : Le programme MACRO.C

```
[1 ] /* Chapitre 7.2 */
[2 ]
[3 ] /* Nom du programme : MACRO.C */
[4 ] /* Exemple de ce qui peut être mal fait */
[5 ]
[6 ] #define FAUX(A) A*A*A              /* faux */
[7 ] #define CUBE(A) (A)*(A)*(A)        /* correct*/
[8 ] #define CARRE(A) (A)*(A)           /* Calcul du carré */
[9 ] #define START 1
[10] #define STOP  9
[11]
[12] main()
[13] {
[14]   int i,constante;
[15]
[16]   constante = 5;
[17]   for (i = START;i <= STOP;i++)
[18]   {
[19]     printf("Le carré de %3d est %4d, et son cube est %6d\n",
[20]     i+constante,CARRE(i+constante), CUBE(i+constante));
[21]     printf("La valeur fausse de %3d est %6d\n",
[22]             i+constante,FAUX(i+constante));
[23]   }
[24] }
```

Ce programme contient, lui aussi, deux constantes symboliques. Sont d'autre part définies trois macros, la première d'entre elles (ligne 6) ayant été volontairement définie de façon à ce que son résultat (le calcul du volume d'un cube) soit erroné dans certaines conditions. L'origine des erreurs éventuelles provient d'un manque de parenthèses.

Les deux macros suivantes renvoient, quant à elles, des calculs systématiquement corrects.

Lancez le programme. Vous verrez que notre première macro détermine des mesures erronées. Considérez le cas où la valeur de la variable *i* est 1, et où celle de la variable *constante* est 5. Pour le calcul du cube, la macro *FAUX(A)* utilise la formule :

```
1+5*1+5*1+5*1=1+5+5+5=16
```

ce qui, de toute évidence est faux. Le calcul correct est obtenu par la macro *CUBE(A)* :

```
(1+5)*(1+5)*(1+5)=6*6*6=216
```

Les parenthèses sont donc utilisées pour regrouper correctement les variables. De même que dans la combinaison d'opérateurs mathématiques et logiques, les parenthèses imposent ici l'ordre de priorité.

7.3. Récapitulation

Les *constantes* sont des valeurs qui ne changent pas pendant le déroulement du programme. Elles contribuent à la lisibilité et à la facilité de maintenance d'un programme. Les constantes symboliques sont celles pour lesquelles une valeur est représentée par un nom. La définition d'une constante doit se trouver au début d'un programme, et être introduite par le mot clé *#define*.

Une macro est une variante spécifique de constante. Les macros peuvent être utilisées comme élément de substitution pour des séries d'instructions. Leur définition est également assurée par *#define*. Un nom de macro est toujours suivi d'une liste de variables écrite entre parenthèses. Lorsque plusieurs variables sont utilisées dans une macro, leur nom doit être séparé par des virgules.

Les définitions de macros et de constantes ne sont pas conclues par un point virgule. En C, l'on écrit habituellement les noms de macros et de constantes en majuscule. Pour définir un ordre de priorité de l'évaluation, on utilise des parenthèses.

Exercices se rapportant au chapitre 7

Exercice 1 :
Ecrivez un programme qui compte de 5 à -5, dans le sens décroissant, et qui renvoie le carré du nombre courant. Vous définirez à cet effet deux constantes pour la valeur supérieure et la valeur inférieure de la liste de comptage, ainsi qu'une macro qui calcule le carré.

Exercice 2 :
Modifiez le programme CELSIUS.C de manière à ce que les valeurs de l'échelle Celsius soient déterminées par l'intermédiaire d'une macro.

8 Chaînes et tableaux

Ce chapitre vous initiera au travail avec les matrices de variables que l'on appelle tableaux (en anglais : arrays). Par la même occasion, vous découvrirez toutes les fonctions standard de manipulation de chaînes de caractères (en anglais : strings). Vous verrez qu'il est possible de programmer le renvoi de parties de chaînes, ainsi que les précautions à prendre lors de la décomposition de tableaux.

Les chaînes et les tableaux sont des structures de données extrêmement importantes en C. Dans ce chapitre, nous nous consacrerons intensivement à la déclaration et au traitement de ces structures de données.

Une *chaîne* est une séquence de caractères qui peut être constituée de lettres, de chiffres et de caractères particuliers.

Dans le langage C, la définition est plus spécifique :

Une chaîne est constituée d'une série de données du type *char*, dont la fin est signalée par la constante '\0'.

Toutes les manipulations de chaînes telles que la comparaison, le renvoi, la modification et la copie sont exécutées en C jusqu'à ce que le symbole de fin de chaîne soit atteint. La pratique s'est répandue de donner à ce caractère le nom de constantes symboliques *NULL*. *NULL* est défini dans le fichier Include STDIO.H, et peut ainsi, par intégration de ce fichier dans le texte du programme, être utilisé en remplacement de la constante '\0'.

8.1. Traitement d'une chaîne

Pour bien comprendre la définition précédente, reprenons un exemple concret de programme. Le programme STRING1.C renvoie, dans un premier temps, une chaîne à l'écran. Ensuite, chaque caractère de cette chaîne est réécrit à l'écran.

Listing 28 : Le programme STRING1.C

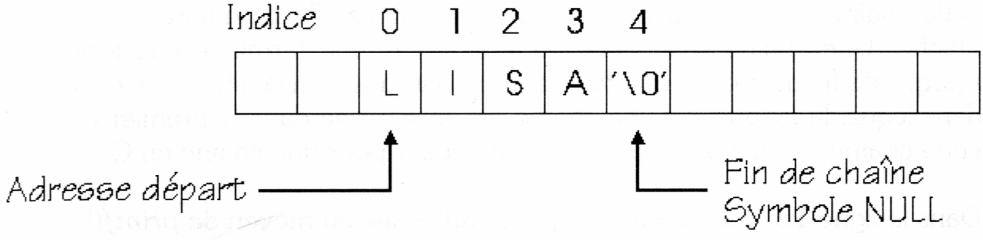
```
[1 ] /* Chapitre 8.1 */
[2 ]
[3 ] /* Nom du programme : STRING1.C */
[4 ] /* Chaîne dont le symbole de fin est NULL */
[5 ]
[6 ]
[7 ] main()
[8 ] {
[9 ]   int x;
[10]   char nom[5];        /* Définition d'une chaîne de 5 caractères */
[11]
[12]   nom[0] = 'L';
[13]   nom[1] = 'I';
[14]   nom[2] = 'S';
[15]   nom[3] = 'A';
[16]   nom[4] = '\0';      /* Null - Fin du texte */
[17]
[18]   printf("Le nom est %s\n",nom); /* renvoi à l'écran de la chaîne */
[19]   for (x=0;x<4;x++)
[20]      printf("La lettre de rang %d est %c\n",x,nom[x]);
                /* Renvoi de chaque caractère de la chaîne */
[21]
[22] }
```

Dans la ligne 10, la déclaration de la variable *nom* est d'un nouveau type. Son type est *char*, et elle est composée d'une série de cinq caractères. D'après la définition énoncée plus haut, il s'agit là d'une chaîne.

La déclaration d'une chaîne s'effectue par conséquent par spécification du nom et de la longueur. La longueur d'une chaîne correspond au nombre des caractères mentionné dans les crochets.

La chaîne *nom* peut être décomposée en ses sous-ensembles, des chaînes partielles, la numérotation (que par la suite il nous arrivera d'appeler indexation) commençant par 0. C'est pourquoi, en plus de la chaîne *nom[5]* sont également définies les chaînes partielles *nom[0]*, *nom[1]*... jusqu'à *nom[4]*.

Si vous êtes néophyte en C, mais aussi programmeur en Pascal, n'oubliez jamais que l'indice (en anglais : subscript) d'une chaîne commence à 0, et que la valeur la plus élevée d'un indice est toujours égale à la longueur de la chaîne, diminuée de 1.



Gestion des chaînes en C

8.1.1. Renvoi de chaînes partielles

La variable *nom[5]* est par conséquent une chaîne qui peut contenir cinq valeurs de type *char*. Etant donné toutefois que le dernier caractère est le symbole *NULL*, seuls quatre caractères significatifs sont à notre disposition. Comme vous pouvez le constater dans les lignes 12 à 16, nous attribuons aux chaînes partielles *nom[0]* à *nom[3]* une lettre unique, et affectons à *nom[4]*, c'est-à-dire au dernier caractère de la chaîne, la constante '\0' qui marque la fin de la chaîne.

Rappelons ici ce que nous avions dit au chapitre 7 : si, dans le programme ci-dessus, vous définissez la constante symbolique NULL, à laquelle vous attribuez la valeur '\0', vous pourrez remplacer l'instruction

```
nom[4] = '\0';
```

par

```
nom[4] = NULL;
```

Vous verrez que la lisibilité du programme est bien plus facile. En particulier, il devient évident que la dernière attribution de valeur (ligne 16) n'est en aucun cas l'affectation du nombre 0.

Revenons au programme dans lequel a été programmé, dans la ligne 18

```
printf("Le nom est %s\n", nom);
```

un renvoi vers l'écran. Cette instruction contient elle aussi une nouveauté. Le descripteur de champ *%s* est utilisé pour l'affichage d'une chaîne. Vous demandez ainsi au programme d'afficher le premier caractère d'une chaîne à la position occupée par le descripteur de champ, et de poursuivre l'affichage de cette chaîne jusqu'à ce que soit atteint le symbole NULL, qui lui, ne sera pas affiché. Une chaîne n'est donc considérée comme terminée que lorsque la marque de fin de chaîne est atteinte. Par conséquent, à la différence du Pascal, dans lequel la longueur d'une chaîne est enregistrée dans le premier octet de cette chaîne, il n'y a aucune limite de longueur pour une chaîne en C.

Dans la ligne 18, vous voyez que pour l'affichage au moyen de *printf()* il suffit de spécifier *nom*, sans indice, en guise de variable.

La boucle *for* programmée dans les lignes 19 et 20 renvoie les chaînes partielles de *nom*. Dans l'instruction

```
printf("La lettre de rang %d est %c\n", x, nom[x]);
```

nous utilisons dans ce but le descripteur de champ *%c* qui sert à renvoyer un caractère de type *char* (y compris les espaces, les sauts de ligne etc.). En guise de variable à afficher, il nous faut spécifier la chaîne partielle avec son indice. Vous noterez que la boucle n'a été programmée que pour les indices 0 à 3, et par conséquent pour les chaînes partielles 1 à 4. Le cinquième caractère (il s'agirait de NULL) ne sera pas renvoyé.

8.1.2. Copier des chaînes

Dans le chapitre précédent, nous avons programmé chaque caractère de la chaîne avec une affectation de valeur. L'exemple suivant vous montre qu'il y a plus simple. Dans ce programme, des chaînes seront copiées, fusionnées et comparées. Vous découvrirez à cette occasion les commandes fondamentales de traitement des chaînes.

Listing 29 : Le programme STRING2.C

```
[1 ] /* Chapitre 8.1.2 */
[2 ]
[3 ] /* Nom du programme : STRING2.C */
[4 ] /*Manipulations de chaînes dans les programmes C */
[5 ] /*Fonctions :      strcpy(); */
[6 ] /*                strcmp(); */
[7 ] /*                strcat(); */
[8 ]
[9 ] main()
[10] {
[11]   char nom1[12],nom2[12],temp[25];
[12]   char titre[20];
[13]
[14]   strcpy(nom1,"Athéna");          /* Copie de la chaîne */
[15]   strcpy(nom2,"Zeus");
[16]   strcpy(titre,"Ceci est un titre");
[17]
[18]   printf("     %s\n\n",titre);    /* Affichage */
[19]   printf("Le nom 1 est %s\n",nom1);
[20]   printf("Le nom 2 est %s\n",nom2);
[21]
[22]   if(strcmp(nom1,nom2)>0)         /* Valeur 1 si nom1 > nom2 */
[23]      strcpy(temp,nom1);
[24]   else
[25]      strcpy(temp,nom2);
[26]
[27]   printf("Du point de vue alphabétique, le nom le plus grand est
            %s\n",temp);
[28]
[29]   strcpy(temp,nom1);
[30]   strcat(temp," et ");
[31]   strcat(temp,nom2);
[32]   printf("Les deux noms sont %s\n",temp);
[33] }
```

Dans un premier temps, nous déclarons quatre variables alphanumériques dans les lignes 11 et 12. Ce sont elles que nous manipulerons ensuite à l'aide de quelques fonctions de manipulation de chaînes. La première de ces fonctions est *strcpy()*, abréviation de "string copy". Cette fonction copie une chaîne jusqu'au caractère NULL dans une autre chaîne, et est intégrée d'origine dans tout compilateur C.

Vous pouvez attribuer à l'instruction

```
strcpy(nom1,"Athéna")
```

le sens suivant :

"attribue à la variable alphanumérique *nom1* la chaîne de caractères Athéna", ou encore "copie tous les caractères de Athéna dans *nom1*."

D'autres chaînes de caractères sont transmises de la même manière aux chaînes *nom2* et *titre*. De par la syntaxe générale de *strcpy()*, vous pouvez considérer que le contenu de la chaîne située à droite de la virgule dans la parenthèse est recopié dans la chaîne placée à gauche de la virgule dans cette parenthèse. Pour cette opération, il n'est pas nécessaire que la chaîne dans laquelle est effectuée la copie ait la même longueur que la chaîne à copier. Une condition est toutefois requise : la première chaîne doit être plus longue d'au moins un caractère que la seconde. Ceci est nécessaire pour le caractère *NULL*.

8.1.3. Comparer des chaînes

La fonction suivante, qui apparaît dans la ligne 22, est *strcmp()*, abréviation de "string compare". Cette fonction compare les chaînes écrites dans les parenthèses de la fonction. Si la première chaîne est supérieure à la deuxième, le résultat de la fonction est 1. Lorsque les deux chaînes sont identiques, le résultat est 0, et lorsque la deuxième chaîne est supérieure à la première, le résultat retourné est -1. Dans la construction *if..else* des lignes 22 à 25, c'est donc la plus "grande" des deux chaînes qui est copiée dans la variable alphanumérique *temp*.

Dans notre cas, Zeus est, du point de vue alphabétique, la chaîne "la plus grande". Notez que le fait qu'une chaîne soit plus grande qu'une autre n'a rien à voir avec sa longueur. C'est le tri dans l'ordre de la table ASCII qui est déterminant. Dans notre cas, la valeur de Z dans la codification ASCII est 90, et est par conséquent supérieure à celle de A dont le code ASCII est 65.

8.1.4. Fusionner des chaînes

La dernière fonction nouvelle dans notre exemple est *strcat()*, abréviation de "string concatenation". Cette fonction permet de fusionner des chaînes. Elle se contente de mettre bout à bout les caractères de la première chaîne mentionnée dans les parenthèses de la fonction, puis ceux de la deuxième chaîne. Cette deuxième chaîne ne sera pas modifiée à cette occasion. Comme vous pouvez le constater en ligne 30, nous copions, dans la dernière partie du programme, la chaîne *nom1* dans la chaîne *temp*. Ensuite, le programme ajoute à *temp* un espace, la chaîne de caractères "et", puis encore un espace, et, dans une

instruction suivante, la chaîne *nom2*. Le résultat de cette manipulation de chaînes

```
Les deux noms sont Athéna et Zeus
```

est renvoyé à l'écran par l'intermédiaire de *printf()*.

Dans la section qui suit, nous nous consacrerons à la structure de données de type array (tableau). Vous apprendrez que les chaînes sont aussi des tableaux, et que grâce à cette structure de données, il est possible de résoudre des tâches très complexes.

8.2. Qu'est-ce qu'un tableau ?

Commençons par la définition :

Un *tableau* est une suite de données qui sont toutes du même type.

Il s'en suit que des chaînes sont des tableaux dont les données sont toutes du type *char*. Dans les tableaux, le type des données peut être extrêmement complexe : nous le verrons d'ailleurs au chapitre 13. Pour l'instant, nous nous contenterons des notions de base. Vous apprendrez à cette occasion tout ce qui concerne la déclaration, l'initialisation et le travail avec des tableaux pluridimensionnels.

8.2.1. Tableau de valeurs entières

Le programme INTARRAY.C qui suit est un premier exemple du travail avec les tableaux de valeurs entières. Il initialise un tableau de type *integer*, puis renvoie celui-ci à l'écran.

Listing 30 : Le programme INTARRAY.C

```
[1 ] /* Chapitre 8.2.1 */
[2 ]
[3 ] /* Nom du programme : INTARRAY.C */
[4 ] /* Exemple de Tableau de valeurs entières */
[5 ]
[6 ] main()
```

```
[7 ] {
[8 ]   int valeurs[12];
[9 ]   int index;
[10]
[11]   for (index = 0;index < 12;index++)
[12]       valeurs[index] = 2 * (index + 4);
[13]
[14]    for (index = 0;index < 12;index++)
[15]       printf("La valeur pour index = %2d est %3d\n",
[16]              index,valeurs[index]);
[17]
[18] }
```

Vous pouvez constater que la déclaration d'un tableau ne diffère en rien de celle d'une chaîne. Le tableau *valeurs[12]* définit 12 variables entières auxquelles l'on peut s'adresser par les noms *valeurs[0]* à *valeurs[11]*. La variable *index* est utilisée dans la boucle *for* des lignes 11 et 12 pour attribuer des valeurs au tableau. Dans ce but, les valeurs de *index* sont d'abord incrémentées de la constante 4, puis multipliées par 2. Dans la boucle suivante, ces valeurs sont alors affichées.

Lorsque vous lancez ce programme, vous constatez que sont exactement calculées 12 valeurs, en commençant par le nombre 0, le plus petit indice.

8.2.2. Initialiser des tableaux

Le programme qui suit met en évidence un tableau de nombres à virgule flottante. Il travaille avec un tableau composé de variables du type *float* (à virgule flottante), qui sont renvoyées structurées sur l'écran. Par la même occasion vous apprendrez comment on peut initialiser un tableau.

Listing 31 : Le programme ARRAY2.C

```
[1 ] /* Chapitre 8.2.2 */
[2 ]
[3 ] /* Nom du programme : ARRAY2.C */
[4 ] /* Initialiser des tableaux - Tableaux avec valeurs de type FLOAT */
[5 ]
[6 ]
[7 ] char nom1[] = "Premier titre du programme";    /* initialisation */
[8 ]
[9 ] main()
[10] {
[11]   int index;
[12]   int nombre1[12];
```

```
[13]    float nombre2[12];
[14]    static char nom2[] = "Deuxième titre du programme";
[15]
[16]    for (index = 0;index < 12;index++)
[17]    {
[18]       nombre1[index] = index + 10;
[19]       nombre2[index] = 12.0 * (index + 7);
[20]    }
[21]
[22]    printf("%s\n",nom1);
[23]    printf("%s\n\n",nom2);
[24]    for (index = 0;index < 12;index++)
[25]       printf("%5d %5d %10.3f\n", index,nombre1[index],nombre2[index]);
[26] }
```

Dans les lignes 7 et 14 vous voyez comment l'on initialise des tableaux. Aucune mention ne figure entre les crochets. Le compilateur déterminera l'espace mémoire nécessaire pour le nombre des caractères attribué, augmenté du symbole de fin de chaîne *NULL*, puis mettra cette plage de mémoire à disposition. La deuxième chaîne est déclarée de type *static*, ce qui signifie que cette variable sera disponible dans le programme jusqu'à sa fin, même si elle a été déclarée comme variable locale (Voir chapitre 6.4).

8.2.3. Tableaux pluridimensionnels

Jusqu'ici, vous n'avez découvert que des tableaux à une dimension. Cela signifie que chaque élément du tableau est caractérisé par un indice unique et spécifique, par l'intermédiaire duquel il est possible de l'adresser. L'exemple de programme DEUXDIM.C qui suit montre qu'il est possible de construire des tableaux pluridimensionnels. Ce programme renvoie pour les nombres 0 à 7 une table d'addition et une table de multiplication.

Listing 32 : Le programme DEUXDIM.C

```
[1 ] /* Chapitre 8.2.3 */
[2 ]
[3 ] /* Nom du programme: DEUXDIM.C */
[4 ] /* Exemple de tableau à deux dimensions */
[5 ]
[6 ]
[7 ] main()
[8 ] {
[9 ]    int i,j;
[10]    int un[8][8],deux[25][12];
[11]
[12]    for (i = 0;i < 8;i++)
[13]       for (j = 0;j < 8;j++)
```

```
[14]            un[i][j]=i*j;          /* Table de multiplication */
[15]
[16]     for (i = 0;i < 25;i++)
[17]        for (j = 0;j < 12;j++)
[18]            deux[i][j] = i + j;    /* Table d'addition        */
[19]
[20]      un[2][6] = deux[24][10]*22;
[21]      deux[2][2] = 5;
[22]      deux[deux[2][2]][deux[2][2]]=177;
[23]       /*deux[5][5] = 177 étant donné que deux[2][2] = 5 !!! */
[24]
[25]
[26]      printf("TABLE DE MULTIPLICATION\n\n");
[27]
[28]      for (i = 0;i < 8;i++)
[29]      {
[30]         for (j = 0;j < 8;j++)
[31]            printf("%5d ",un[i][j]);
[32]            printf("\n");        /* nouvelle ligne pour i = i + 1 */
[33]      }
[34]      printf("\n\n");
[35]
[36]      printf("T A B L E D' A D D I T I O N\n\n");
[37]      for (i = 0;i < 8;i++)
[38]      {
[39]         for (j = 0;j < 8;j++)
[40]            printf("%5d ",deux[i][j]);
[41]            printf("\n");     /* nouvelle ligne pour i = i + 1 */
[42]      }
[43] }
```

Nous déclarons ici, dans la ligne 10, deux tableaux bidimensionnels. Le *tableau un[8][8]* est composé de 8*8 éléments. Nous allons voir d'un peu plus près le principe des tableaux pluridimensionnels à l'exemple du tableau *un*.

Ce tableau regroupe 64 variables, à commencer par *un[0][0]*, la dernière étant *un[7][7]*. La boucle à double imbrication programmée dans les lignes 12 à 14 remplit le tableau avec une table de multiplication. La variable *un[0][0]* renferme la valeur 0*0, *un[0][1]* renferme 0*1 et ainsi de suite. La dernière variable de ce champ, c'est-à-dire *un[7][7]* reçoit pour finir la valeur 7*7. Dans les lignes 16 à 18, les variables du tableau *deux* se voient attribués les résultats d'une table d'addition. Suite à cela, nous voyons trois instructions qui mettent en évidence que des valeurs quelconques peuvent être affectées aux éléments du tableau *un*, et que toute expression valable peut être utilisée comme indice. C'est ainsi que, dans la ligne 20, la variable *un[2][6]* reçoit par l'intermédiaire de l'instruction

```
un[2][6] = deux[24][10]*22;
```

la valeur de la variable *deux[24][10]*, préalablement multipliée par 22. Ensuite, dans la ligne 21, la valeur 5 est affectée à la variable *un[2][2]*.

La ligne 22 du programme

```
Deux[deux[2][2]][deux[2][2]]=177;
```

montre que la valeur d'un élément du tableau peut être utilisée comme indice pour un autre élément de tableau. En raison de l'affectation précédente (ligne 21), cette instruction pourrait aussi être écrite sous la forme :

```
un[5][5] = 177
```

Cela nous permet d'énoncer la règle suivante :

Peut être utilisée comme indice d'un élément de tableau toute expression qui remplit les conditions suivantes :

① La valeur doit être du type *int* ou *char*.

② La valeur doit appartenir au domaine de définition du tableau.

Pour le tableau *un[8][8]*, peuvent donc être utilisés des nombres entiers appartenant au domaine de définition compris entre 0 et 7. Rappelons ici que la numérotation des indices commence toujours par 0.

Voici un deuxième programme, DEUXDIM2.C, qui permet de mieux mettre en évidence les principes décrits plus haut.

Listing 33 : Le programme DEUXDIM2.C

```
[1 ] /* Chapitre 8.2.3 */
[2 ]
[3 ] /* Nom du programme : DEUXDIM2.C */
[4 ] /* Autre exemple de programme avec tableaux bidimensionnels */
[5 ]
[6 ] #include "stdio.h"
[7 ] attends()
[8 ] {
[9 ]    char c;
[10]
[11]    printf("Touche quelconque pour continuer .......");
[10]    c=getch();
[11]    printf("\n");
[12] }
[13]
[14] main()
[15] {
[16]    int i,j,n,grand[8][8],plusgrand[25][12];
[17]
[18]    for (i = 0;i < 8;i++)
```

```
[19]        for (j = 0;j < 8;j++)
[20]            grand[i][j] = i*j;      /* Table de multiplication */
[21]
[22]    printf("T A B L E  D E  M U L T I P L I C A T I O N\n");
[23]   for (i = 0,n=0;i < 8;i++)
[24]     for (j = 0;j < 8;j++)
[25]     {
[26]        printf("La valeur de grand(%d,%d) est \t %d\n",
[27]                 i,j,grand[i][j]);
[28]        n++;
[29]        if (n == 23)
[30]        {
[31]          attends();
[32]          n=0;
[33]          printf("TABLE DE MULTIPLICATION \n");
[34]        }
[35]      }
[36]   attends();
[37]   for (i = 0;i < 25;i++)
[38]      for (j = 0;j < 12;j++)
[39]          plusgrand[i][j]=i+j;    /*Table d'addition*/
[40]   printf("T A B L E  D' A D D I T I O N\n");
[41]   for (i = 0,n=0;i < 8;i++)
[42]      for (j = 0;j < 8;j++)
[43]          {
[44]              printf("La valeur de plusgrand(%d,%d) est \t %d\n",
[45]                      i,j,plusgrand[i][j]);
[46]              n++;
[47]              if (n == 23)
[48]              {
[49]                  attends();
[50]                  n=0;
[51]                  printf("TABLE D'ADDITION \n");
[52]              }
[53]          }
[57] }
```

Lancez le programme. Dans un souci d'amélioration de la clarté de l'affichage, il utilise la fonction *attends()*, dont vous pouvez étudier les effets et la structure après avoir vu le chapitre 11. Grâce à un affichage suffisamment explicite, nous allons pouvoir clarifier le principe des tableaux pluridimensionnels.

L'affichage renvoyé par le programme DEUXDIM2.C est le suivant

```
T A B L E  D E  M U L T I P L I C A T I O N
La valeur de grand(0,0) est        0
La valeur de grand(0,1) est        0
La valeur de grand(0,2) est        0
La valeur de grand(0,3) est        0
La valeur de grand(0,4) est        0
La valeur de grand(0,5) est        0
La valeur de grand(0,6) est        0
```

```
La valeur de grand(0,7) est        0
La valeur de grand(1,0) est        0
La valeur de grand(1,1) est        1
La valeur de grand(1,2) est        2
La valeur de grand(1,3) est        3
La valeur de grand(1,4) est        4
La valeur de grand(1,5) est        5
La valeur de grand(1,6) est        6
La valeur de grand(1,7) est        7
La valeur de grand(2,0) est        0
La valeur de grand(2,1) est        2
La valeur de grand(2,2) est        4
La valeur de grand(2,3) est        6
La valeur de grand(2,4) est        8
La valeur de grand(2,5) est        10
La valeur de grand(2,6) est        12
Touche quelconque pour continuer...
```

Le programme de démonstration suivant récapitule les procédés de traitement de *strings* (chaînes alphanumériques), ainsi que la technique de récursion décrite au chapitre 6.

Listing 34 : Le programme RETOUR.C

```
[1 ] /* Chapitre 8.2.3 */
[2 ]
[3 ] /* Nom du programme : RETOUR.C */
[4 ] /* Autre exemple de programmation récursive */
[5 ]
[6 ]
[7 ] #include "stdio.h"  /* Prototypes standard pour entrées/sorties */
[8 ] #include "string.h" /* Prototypes pour opérations sur des chaînes */
[9 ]
[10] void avant_et_arriere(char line_of_char[],int index);
[11]
[10] main()
[11] { /* Début main() */
[12]   char chaine_caracteres[80];
[13]   char serie_nombres[12];
[14]   int index = 0;
[15]
[16]   strcpy(line_of_char,"\nCeci est une chaîne.\n");
[17]   strcpy(serie_nombres,"\n0123456789\n");
[18]   avant_et_arriere(chaine_caracteres,index);
[19]   avant_et_arriere(serie_nombres,index);
[20] } /* Fin de main() */
[21]
[22] void avant_et_arriere(char line_of_char[],int index)
[23] {
[24]   if (line_of_char[index]) /* jusqu'à la fin de la chaîne */
[25]   {
[26]      printf("%c",line_of_char[index]);
[27]      index++;
```

```
[28]      avant_et_arriere(line_of_char,index); /* Récursion */
[29]    }
[30]   printf("%c",line_of_char[index]); /* Fin de la Récursion */
[31] } /* Fin de la fonction */
```

Dans ce listing, il est une fonction qui mérite une attention particulière :

```
void avant_et_arriere(char line_of_char[],int index)
```

Dans la ligne 28, cette fonction s'appelle elle-même (récursion) aussi longtemps que l'expression

```
line_of_char[index]
```

est vraie. Et celle-ci est vraie tant que la fin de la chaîne n'est pas atteinte. Ce n'est qu'ensuite que la dernière instruction de la fonction (ligne 30) sera exécutée : la chaîne transmise sera une fois encore retournée à l'écran caractère par caractère, mais cette fois, dans l'ordre inverse.

8.3. Contrôle logique à l'aide du débogueur (II)

Utilisons cet exemple de programme pour une nouvelle démonstration des techniques de travail avec le débogueur.

Positionnez le curseur dans la ligne 27, et placez-y un point d'arrêt avec <Ctrl>+<F8> (sous Turbo C et C++) ou avec <F9> (dans Quick C 2.5). Répétez cette opération dans les lignes 28 et 31. Placez ensuite le curseur sur la variable *index*, dans la ligne 28, et appuyez sur les touches <Ctrl>+<F7> (Turbo C 2.0 et C++) ou, dans Quick C 2.5, activez l'option *Débogage/Visualiser la variable*. Vous venez d'attribuer à cette variable la caractéristique *Watch*, et pourrez ainsi, par la suite, visualiser son contenu pendant l'exécution du programme. Positionnez enfin le curseur dans la ligne 30 sur la variable *line_of_char[index]*, et répétez l'opération.

Vous venez de marquer trois lignes auxquelles le programme s'arrêtera pendant son exécution. De plus, vous pouvez suivre à l'écran pendant le déroulement du programme, les variations des variables *index et line_of_char[index]*. Dans ce but, appuyez maintenant sur <Ctrl>+<F9> (Turbo C et C++) ou sur <F5> (Quick C 2.5). Le programme s'exécute jusqu'à la première ligne que nous avons

marquée et s'arrête. Ce processus se répète à chaque nouvelle pression sur <Ctrl>+<F9> (respectivement <F5>), et, à chaque point d'arrêt que nous avons placé, nous pouvons voir le contenu des variables *index* et *line_of_char[index]*. A cet effet, il faut, dans l'environnement de développement de C++, ouvrir la fenêtre de visualisation (Watch) dans la zone inférieure de l'écran, par l'intermédiaire de la commande *Window/Watch*. Dans Turbo C 2.0, cela n'est pas nécessaire. Quick C 2.5 vous montre les valeurs des variables dans la partie supérieure de la fenêtre.

Si vous lancez le programme de démonstration précédent en mode Step-Over, vous pouvez constater que la valeur de *index* croît, et que dans le même sens de variation, des caractères de la chaîne alphanumérique

```
Ceci est une chaîne.
```

sont progressivement ajoutés au tableau *line_of_char*. Lorsque *index* prend la valeur 22, le caractère *NULL* marque la fin de la chaîne. La condition d'appel récursif de la fonction (ligne 24) n'est alors plus remplie, et le programme se met à traiter la pile (Stack). Appuyez maintenant sur la combinaison de touches <Ctrl>+<F3> (Turbo C et C++), ou cliquez sur l'option *Débogage/Appels* (Quick C). Le contenu actuel de la pile est visualisé à l'écran.

Appuyez plusieurs fois de suite sur <F8>. Vous voyez que la valeur de *index* diminue de 1 après chaque étape du programme. Comme vous le savez, la pile est maintenant traitée suivant le principe LIFO, et la chaîne précédemment attribuée sera donc affichée à l'écran dans l'ordre inverse de ses caractères. Afin de contrôler la sortie, appuyez, après quelques étapes du programme, sur la combinaison de touches <Alt>+<F5> (Turbo C et C++) ou sur <F4> (Quick C 2.5). Vous voyez l'écran utilisateur avec un affichage encore incomplet renvoyé par notre programme.

8.4. Récapitulation

Les chaînes (strings) sont des séries de caractères dont la fin est marquée par la constante '\0'. Chaque caractère d'une chaîne peut être adressé par l'intermédiaire de son indice. Il s'agit là du numéro du caractère dans la chaîne alphanumérique, la numérotation commençant toujours par 0. La longueur d'une chaîne et l'indice sont toujours écrits entre crochets. Le descripteur de champ pour le renvoi d'une chaîne est *%s*. Le renvoi des caractères individuels d'une chaîne est obtenu par *%c*.

Les fonctions dont on dispose pour traiter les chaînes alphanumériques sont : *strcpy()* (copier), *strcmp()* (comparer) et *strcat()* (fusionner). Ces fonctions font partie des fonctions standard d'un compilateur C.

Les tableaux (arrays) sont des champs de données dont tous les éléments sont du même type. Les strings (chaînes alphanumériques) sont donc nécessairement des tableaux (arrays) dont toutes les données sont de type *char*. Toute variable ou constante de type *int* ou *char* appartenant au domaine de définition du tableau peut être utilisée comme indice dans ce tableau. La valeur la plus faible d'un indice est 0. Sa valeur la plus élevée est égale à la longueur du champ diminuée de 1. Chaque élément d'un tableau peut être considéré comme une variable autonome.

La définition de tableaux pluridimensionnels est possible. Dans ce but, il faut mentionner entre crochets le nombre des éléments pour chaque dimension.

Exercices se rapportant au chapitre 8

Exercice 1 :
Ecrivez un programme qui détermine la longueur d'une chaîne et la renvoie à l'écran. Vous utiliserez à cet effet une fonction définie de la manière suivante :

```
int longueur_string(char strg[])
```

Exercice 2 :
Ecrivez un programme qui décompose le chemin

```
C:\DAT\TXTDAT\LIVRE\
```

en ses divers sous-répertoires et les renvoie à l'écran.

Exercice 3 :
Ecrivez un programme qui détermine la longueur d'un mot que l'utilisateur devra saisir.

9 Transmission de données à des fonctions

Dans ce chapitre vous verrez comment transmettre des données à une fonction de manière à ce que cette fonction puisse modifier directement la valeur de ces variables.

Dans le chapitre 6, vous avez découvert deux techniques qui renvoient les données provenant d'une fonction vers le programme appelant. Dans les paragraphes qui suivent, vous découvrirez une nouvelle technique, très efficace, qui produit le même effet. Nous utiliserons comme exemple le programme FRETOUR.C.

Listing 35 : Le programme FRETOUR.C

```
[1 ] /* Chapitre 9 */
[2 ]
[3 ] /* Nom du programme: FRETOUR.C */
[4 ] /* Exemple de renvoi de tableaux par des fonctions */
[5 ]
[6 ] main()
[7 ] {
[8 ]   int index;
[9 ]   int matrice[20];
[10]
[11]   for (index = 0;index < 20;index++)     /* Créer des données */
[12]      matrice[index] = index + 1;
[13]
[14]   for (index = 0;index < 5;index++)      /* Renvoi de données */
[15]      printf("Départ : matrice[%d] = %d\n",index,matrice[index]);
[16]
[17]   agis(matrice);     /* Cette fonction modifie le tableau */
[18]
```

```
[19]   for (index = 0;index < 5;index++)      /* Renvoi de matrice */
[20]      printf("Retour de la fonction  matrice[%d] = %d\n",
[21]              index,matrice[index]);
[22] }
[23]
[24] int agis(int liste[])  /* montre le renvoi des valeurs du tableau */
[25] {
[26]    int i;
[27]
[28]    for (i = 0;i < 5;i++)      /* Imprime le tableau original */
[29]       printf("Avant : matrice[%d] = %d\n",i,liste[i]);
[30]
[31]    for (i = 0;i < 20;i++)     /* ajoute 10 à toutes les valeurs */
[32]       liste[i] += 10;
[33]    for (i = 0;i < 5;i++)      /* Imprime le tableau modifié */
[34]       printf("Après : matrice[%d] = %d\n",i,liste[i]);
[35] }
```

9.1. Modifier des données externes

Nous définissons ici le tableau *matrice* avec 20 variables auxquelles nous attribuons des valeurs initiales par l'intermédiaire d'une boucle dans les lignes 11 et 12. Pour pouvoir contrôler l'affectation des cinq premières valeurs aux variables *matrice[0]* à *matrice[4]*, nous ordonnons leur affichage à l'écran. Ensuite, nous activons la fonction *agis()* à laquelle nous transmettons le tableau *matrice* dans les accolades de la fonction. La fonction appelée traite ce tableau sous le nom *liste*. Grâce à la déclaration de type qui figure en ligne 24, la fonction "sait" que, lors de son activation, c'est un tableau de variables entières qui lui est transmis. Il n'est pas nécessaire de spécifier le nombre d'éléments que contient ce tableau. La règle générale qui s'applique dans ce cas est :

Une fonction traite un tableau jusqu'à ce qu'elle reconnaisse la marque de fin de données.

Jusque ici, il n'y a aucune différence par rapport aux fonctions que nous avions programmées précédemment, si ce n'est que le nombre des données transmises est supérieur. Dans les lignes 28 et 29, nous demandons à nouveau le renvoi à l'écran des cinq premières valeurs. Il est ainsi possible de suivre les modifications que la fonction apporte aux données. Le programme ajoute ensuite la constante 10 aux valeurs transmises, et renvoie une seconde fois les cinq premières valeurs à l'écran. Nous revenons alors à *main()* où, dans les lignes 19 à 21, les cinq premières valeurs modifiées sont renvoyées à l'écran.

Lancez le programme, et observez le résultat produit.

Voici l'affichage renvoyé par le programme FRETOUR.C :

```
Départ : matrice[0] = 1
Départ : matrice[1] = 2
Départ : matrice[2] = 3
Départ : matrice[3] = 4
Départ : matrice[4] = 5
Avant : matrice[0] = 1
Avant : matrice[1] = 2
Avant : matrice[2] = 3
Avant : matrice[3] = 4
Avant : matrice[4] = 5
Après : matrice[0] = 11
Après : matrice[1] = 12
Après : matrice[2] = 13
Après : matrice[3] = 14
Après : matrice[4] = 15
Retour de la fonction  matrice[0] = 11
Retour de la fonction  matrice[1] = 12
Retour de la fonction  matrice[2] = 13
Retour de la fonction  matrice[3] = 14
Retour de la fonction  matrice[4] = 15
```

Comme vous pouvez le voir, les valeurs de notre tableau ont également été modifiées dans *main()*. Vous avez donc modifié et retourné à la partie appelante du programme plus qu'une valeur. Comment en sommes-nous arrivés là ?

Dans le chapitre 6, nous avions vu que le programme appelant transmet toujours une copie des valeurs transmises à la fonction appelée, et que par conséquent les valeurs initiales restent inchangées. Cette règle ne s'applique toutefois pas aux tableaux (arrays). Dans ce cas, ce sont les adresses des valeurs originales qui sont transmises, ces valeurs pouvant ensuite être modifiées par la fonction appelée. Il en résulte que ces modifications seront ensuite valables également dans la partie appelante du programme.

Le traitement des tableaux (arrays) est par conséquent différent de celui des types de données simples tels que *int*, *char* ou *float*.

La raison pour laquelle il en est ainsi sera étudiée dans le chapitre suivant.

9.2. Récapitulation

Grâce aux tableaux, il est possible qu'une fonction retourne à la partie appelante d'un programme plus qu'une unique valeur modifiée. Cela provient du fait que, dans ce cas, ce n'est pas une copie qui est transmise en guise d'argument, mais les adresses des variables à traiter, la fonction pouvant ainsi accéder directement aux données originales.

Exercices se rapportant au chapitre 9

Exercice 1 :
Ecrivez un programme qui initialise le tableau d'entiers *champint[]* avec les valeurs

```
22,100,1,19,250,8,4,20,30,0
```

puis qui la trie à l'aide de la fonction

```
champ_tri(int tableau_tri[])
```

dans l'ordre croissant, et enfin qui renvoie le résultat à l'écran à l'aide de la fonction

```
imprime(int tableau_tri[])
```

Exercice 2 :
Modifiez le programme ci-dessus de manière à ajouter l'argument *int sens* dans la fonction de tri, dans

```
champ_tri(int tableau_tri[],int sens)
```

Lorsque l'argument *sens* reçoit la valeur 1, le tri devra être réalisé dans l'ordre croissant, en commençant par la valeur la plus petite. Lorsque sa valeur sera 2, le tri devra être décroissant, à commencer par la valeur la plus grande.

10 Pointeurs

Dans ce chapitre, vous découvrirez l'utilisation des pointeurs, concept fondamental en C. Grâce à des programmes de démonstration, nous présenterons l'utilisation de l'opérateur d'adressage, la déclaration et l'initialisation de pointeurs, et enfin l'arithmétique spécifique aux pointeurs. Nous aborderons également la relation entre tableaux et pointeurs, et vous apprendrez à initialiser des tableaux de pointeurs (en anglais : pointerarrays). Un dernier exemple enfin traitera de l'utilisation de ces notions dans la pratique.

Voyons d'abord ce qu'est un pointeur.

Un pointeur est une adresse.

Un pointeur n'est pas une variable : il contient l'adresse d'une variable. L'adresse d'une variable est le numéro de la cellule vive dans laquelle cette variable est mémorisée. La meilleure façon d'aborder un nouveau sujet reste l'exemple : le programme POINTER.C nous apporte une démonstration de l'utilisation de pointeurs.

Listing 36 : Le programme POINTER.C

```
[1 ] /* Chapitre 10 */
[2 ]
[3 ] /* Nom du programme : POINTER.C */
[4 ] /* Premier exemple d'utilisation de POINTEURS */
[5 ]
[6 ] main()
[7 ] {
```

```
[8 ]   int index,*pt1,*pt2;
[9 ]
[10]   index = 39;              /* valeur numérique quelconque */
[11]   pt1 = &index;            /* l'adresse de index */
[12]   pt2 = pt1;
[13]   printf("La valeur de index est           \t %d \n",index);
[14]   printf("La valeur de *pt1  est           \t %d \n",*pt1);
[15]   printf("La valeur de *pt2  est           \t %d \n\n",*pt2);
[16]
[17]   *pt1 = 13;               /* la valeur de index change */
[18]
[19]   printf("La valeur de index est maintenant  \t %d \n",index);
[20]   printf("La valeur de *pt1  est également   \t %d \n",*pt1);
[21]   printf("Et celle de *pt2   est             \t %d \n",*pt2);
[22]
[23] }
```

10.1. Deux règles importantes

Ne tenez pas compte, dans un premier temps, des déclarations de types de *index*, **pt1* et **pt2*, dans la ligne 8. Nous reviendrons plus loin sur la signification de l'astérisque placé devant les noms. L'attribution de valeur

```
index = 39;
```

qui figure en ligne 10 vous est déjà connue. Par contre, la syntaxe de la ligne 11 est nouvelle :

```
pt1 = &index;
```

Cette affectation signifie qu'à la variable *pt1* est attribuée l'adresse de la variable *index* à l'aide de l'opérateur *&*. L'adresse d'une variable est le numéro de la cellule de la mémoire principale dans laquelle cette variable est mémorisée. Dans notre exemple, *pt1* et *pt2* sont des pointeurs, et *index* est une variable. Avant d'expliquer pourquoi il en est ainsi, voici deux règles importantes qui nous aideront à mieux comprendre la signification et l'utilisation des pointeurs.

Une variable à laquelle est attribuée l'adresse d'une autre variable par l'intermédiaire de l'opérateur & (appelé Ampersand en anglais) est un pointeur dirigé sur cette variable. L'instruction

```
pt1 = &index
```

signifie par conséquent : "attribue au pointeur *pt1* l'adresse de la variable *index*".

Le signe & est l'opérateur d'adressage, et doit être placé devant le nom de la variable.

En C, un pointeur introduit par un astérisque est appelé indirection, et concerne la variable sur laquelle est dirigé le pointeur.

Il s'en suit que la signification de l'instruction

```
*pt1 = 13
```

qui figure à la ligne 17 est : "attribue la valeur 13 à la variable sur laquelle est dirigé le pointeur *pt1*".

Pour plus de simplicité, il convient surtout de retenir :

& est toujours une adresse

*est toujours une indication concernant une valeur mémorisée.

10.2. Deux pointeurs pour une variable

Avec *pt1* et *pt2*, nous avons donc déclaré des pointeurs, et ceux-ci sont des variables qui ne renferment pas la valeur d'une variable mais l'adresse d'une variable. Ils peuvent par conséquent être utilisés comme pointeurs dirigés sur une variable. Ainsi, lorsque vous attribuez à *pt2* la valeur de *pt1* par l'instruction

```
pt2 = pt1;
```

pt2 sera dirigé sur la même variable que *pt1*, c'est-à-dire sur *index*, et par conséquent sur la valeur 39. Nous avons en effet attribué la valeur 39 à cette variable dans la ligne 10.

Vous pouvez donc modifier la valeur de *index* soit en utilisant le nom de la variable, soit en faisant appel à un pointeur dirigé sur *index*. Cette technique est utilisée dans la ligne 17 dans l'instruction

```
*pt1 = 13;
```

D'après la deuxième règle que nous avons énoncée dans ce chapitre, l'astérisque placé devant *pt1* signifie que l'affectation d'une valeur concerne la variable dont

l'adresse est mémorisée dans le pointeur, et non le pointeur lui-même. Cette instruction est donc suivie du même effet que

```
index = 13;
```

A tout endroit du programme où vous pourriez utiliser la variable *index*, vous avez également le droit d'employer le nom **pt1*, si *pt1* est un pointeur dirigé sur *index*. Tant que vous n'affectez pas à ce pointeur l'adresse d'une autre variable, ces deux désignations sont identiques.

Pour mieux mettre en évidence le principe des pointeurs, nous avons introduit dans le programme un deuxième pointeur : *pt2*. Tant qu'aucune adresse n'a été affectée à *pt2*, ce pointeur n'est dirigé sur rien, il renferme simplement une valeur aléatoire qui, avant le lancement du programme, se trouvait déjà dans la cellule de mémoire actuellement affectée au pointeur. Dans l'instruction

```
pt2 = pt1;
```

l'adresse pt1 est affectée à *pt2*, les deux pointeurs étant maintenant dirigés sur *index*. Dans ces conditions, vous pourriez dès lors remplacer le nom de variable *index* par les noms de pointeurs **pt2* ou **pt1*. Les fonctions *printf()* des lignes 19 à 21 de notre programme de démonstration en apportent la preuve.

Nous disposons maintenant d'une variable et de deux pointeurs. Notez bien que par l'instruction

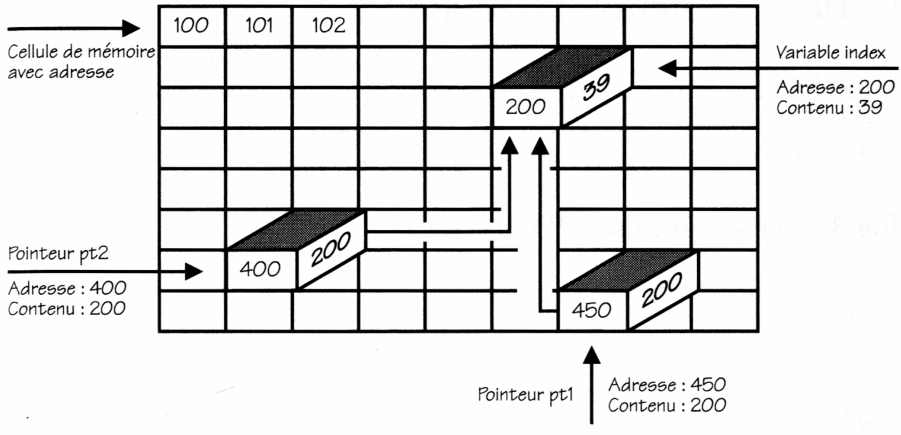
```
*pt1 = 13;
```

ce n'est réellement qu'une valeur (celle de *index*) qui est modifiée. La valeur des deux pointeurs (l'adresse de *index*) ne change pas.

Le concept des pointeurs est délicat. Toutefois, étant donné leur usage intensif en C, l'effort de s'y consacrer intensivement mérite d'être fait. N'hésitez donc pas à consacrer beaucoup de temps à ce chapitre, afin de vous assurer d'avoir bien tout compris.

La figure qui suit présente graphiquement le principe d'un pointeur, en utilisant les données de notre programme de démonstration. Elle vous montre qu'il est indispensable de faire la distinction entre adresse d'une variable et contenu d'une variable.

Par conséquent, si le contenu des pointeurs *pt1* et *pt2* est l'adresse de la variable *index*, il est possible d'accéder à *index* en passant par ces pointeurs.



La relation entre pointeurs et variables

10.3. Déclarer des pointeurs

Venons-en maintenant à la déclaration des pointeurs. Observez à cet effet la partie déclaration de notre programme. Dans la ligne 8, vous y trouvez la déclaration

```
int *pt1, *pt2;
```

Celle-ci signifie que *pt1* et *pt2* sont des pointeurs, et que les emplacements de la mémoire vers lesquels ces pointeurs sont dirigés sont de type *int*. Ceci nous permet d'énoncer une troisième règle.

Les pointeurs doivent être déclarés pour un type de données bien précis.

Ainsi, un pointeur dirigé sur des variables entières (*integer*) ne peut pas être utilisé pour pointer sur une variable de type *float*.

10.4. Pointeurs et tableaux (arrays)

Dans ce chapitre, vous avez déjà découvert de nombreuses nouvelles notions, et il en viendra encore d'autres. Le programme de démonstration POINTER2.C qui suit constitue une nouvelle étape dans ce domaine.

Listing 37 : Le programme POINTER2.C

```
[1 ] /* Chapitre 10.4 */
[2 ]
[3 ] /* Nom du programme : POINTER2.C */
[4 ] /* Traitement de tableaux à l'aide de pointeurs */
[5 ]
[6 ] main()
[7 ] {
[8 ]   char strg[40],*ailleurs,un,deux;
[9 ]   int *pt,liste[100],index,compteur;
[10]
[11]   strcpy(strg,"Ceci est une chaîne de caractères.");
[12]   printf("\t\t\%s\n",strg);
[13]
[14]   un = strg[0];            /* un et deux sont identiques */
[15]   deux = *strg;
[16]   printf("Le premier renvoi est %c %c\n",un,deux);
[17]
[18]   un = strg[13];           /* un et deux sont identiques */
[19]   deux = *(strg+13);
[20]   printf("Le deuxième renvoi est %c %c\n",un, deux);
[21]   ailleurs = strg+10;      /* strg+10 est identique à strg[10] */
[22]   printf("Le troisième renvoi est %c\n",strg[10]);
[23]   printf("Le quatrième renvoi est %c\n",*ailleurs);
[24]
[25]   for (index = 0;index < 100;index++)
[26]      liste[index] = index + 100;
[27]   pt = liste + 27;
[28]   printf("Le cinquième renvoi est %d\n",liste[27]);
[29]   printf("Le sixième renvoi est %d\n",*pt);
[30]
[31]   /* Petite répétition à propos de la boucle WHILE */
[32]   index = 0;
[33]   compteur = 0;
[34]   while (strg[index])
[35]   {
[36]      printf("%c  ",strg[index]);
[37]      compteur+=4;
[38]      if (compteur > 76)
[39]      {
[40]         printf("\n");
[41]         compteur = 0;
[42]      }
[43]   index+=1;
```

```
[44]    }
[45] }
```

Dans la ligne 8 de ce programme sont déclarées quatre variables de type *char*, et dans la ligne 9, quatre variables entières (*int*). Parmi celles-ci se trouvent deux pointeurs. Le premier s'appelle *ailleurs*, et est dirigé sur une variable de type *char*. Le second s'appelle *pt* et pointe sur une variable entière. Sont d'autre part déclarés deux tableaux, *strg* et *liste*, qui nous permettront de mettre en évidence la relation entre les pointeurs et les tableaux.

La variable *strg[40]* est une chaîne dont nous pouvons décrire maintenant avec plus de précision la signification.

En C, une chaîne n'est rien de plus qu'un pointeur dirigé sur l'origine d'un tableau (*array*) de valeurs du type *char*.

Comme vous pouvez le constater dans la ligne 11, le programme copie d'abord une constante alphanumérique dans le tableau *strg[40]*, qui est immédiatement renvoyée à l'écran. Ensuite, la valeur de *strg[0]* est attribuée à la variable *un*, qui reçoit donc le premier caractère de notre constante alphanumérique. A la variable *deux* de type *char* est également affectée la première valeur de la chaîne. Par conséquent, le contenu de la variable *deux* est maintenant identique au premier caractère de la chaîne *strg[40]*.

L'instruction

```
deux = *strg
```

peut être lue de la façon suivante :

"attribue à la variable *deux* la valeur qui se trouve dans la cellule de mémoire sur laquelle est dirigé le pointeur *strg*". Or, *strg* pointe sur l'origine de la chaîne *strg[40]* !

Si vous poursuivez la lecture du programme, vous voyez deux autres instructions qui montrent qu'un pointeur peut également être indexé. Ainsi, dans la ligne 18,

```
un = strg[13];
```

affecte à la variable "*un*" le quatorzième caractère de la chaîne. Le même résultat est obtenu pour la variable "*deux*" avec l'instruction qui suit, sauf que cette fois c'est un pointeur qui est utilisé pour définir l'affectation.

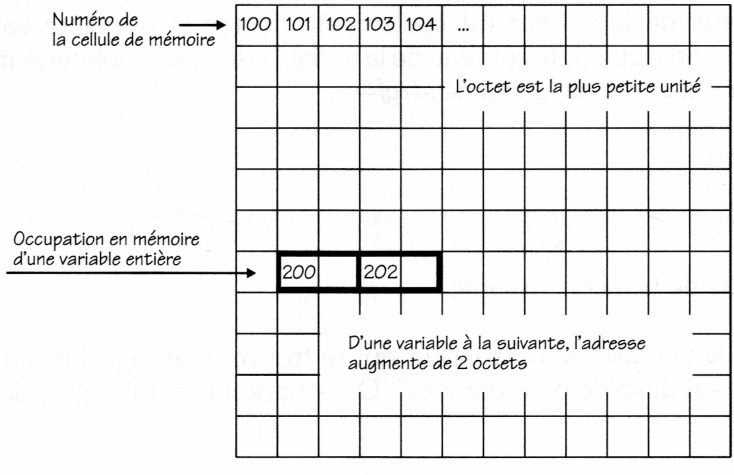
L'on peut donc indexer un pointeur pour accéder aux cellules de mémoire qui suivent.

Par conséquent, l'expression

```
deux = *(strg+13)
```

signifie : "attribue à la variable *deux* la valeur qui correspond à la treizième cellule de mémoire qui suit la valeur de l'origine de la chaîne *strg*."

Le langage de programmation C se charge ici automatiquement d'une indexation correcte. Etant donné que chaque pointeur est déclaré pour un certain type de données, le compilateur "sait" de combien d'octets il faut augmenter l'adresse concernée, et ce pour chaque pointeur. Dans notre cas, il s'agit de données du type char, l'incrémentation de l'index devant alors être d'un octet. Pour les variables entières (integer), le pas de l'incrémentation est de deux octets. La figure suivante met graphiquement ce principe en évidence.



L'arithmétique des pointeurs

Poursuivons l'analyse du programme. Etant donné que "*ailleurs*" a été déclaré comme pointeur, nous savons que l'instruction

```
ailleurs = strg+10;
```

qui se trouve dans la ligne 21 affecte au pointeur l'adresse du onzième élément de notre chaîne de caractères, puisque *strg* est l'adresse du premier élément. C'est pourquoi l'instruction *printf* qui suit renvoie les mêmes lettres à l'écran.

Vous pouvez ajouter ou soustraire des valeurs à un pointeur. L'addition signifie que le pointeur sera ensuite dirigé sur une cellule de mémoire qui suit la position initiale du pointeur. La soustraction, quant à elle, procède à une "marche arrière" en mémoire. Le fait d'ajouter, de multiplier ou de diviser des pointeurs entre eux ne présente par contre aucun intérêt, et fournit des résultats insensés.

10.5. Pointeur dirigé sur des valeurs entières

Lorsque vous poursuivez la lecture du programme, vous voyez qu'au tableau *liste*, (lignes 25 et 26) sont affectées des valeurs comprises entre 100 et 199 par l'intermédiaire d'une boucle *for*. Ensuite, par l'intermédiaire de l'instruction

```
pt = liste[27]
```

un pointeur est dirigé sur le vingt huitième (!) élément de la liste de nombres entiers, et cette valeur est renvoyée deux fois à l'écran. Le premier affichage est dû à la variable *liste[27]*, et le deuxième est obtenu par l'intermédiaire du pointeur *pt* dirigé sur cette variable. La boucle *while* répartie sur les lignes 34 à 44 est prévue pour servir de répétition.

10.6. Transmission de données à des fonctions à l'aide de pointeurs

L'exemple de programme qui suit vous montre que, à l'aide de pointeurs, l'on peut retourner à la partie appelante d'un programme des valeurs modifiées dans une fonction appelée. C'est là l'un des points forts de cette technique de programmation.

Listing 38 : Le programme DEUXVOIE.C

```
[1 ] /* Chapitre 10.6 */
[2 ]
[3 ] /* Nom du programme : DEUXVOIE.C */
[4 ] /* Exemple de transmission de données de fonctions */
[5 ] /* à l'aide de pointeurs */
[6 ] main()
[7 ] {
[8 ]   int var1,var2;
[9 ]
[10]   var1 = 100;
[11]   var2 = 101;
[12]   printf("Les valeurs initiales de var1 et var2 sont  %d %d\n",
[13]          var1,var2);
[14]
[15]                        /* Lorsque nous activons "prog" */
[16]   prog(var1,&var2);    /* nous "emmenons" la valeur de  var1  */
[17]                        /* et l'adresse de var2 */
[18]
[19]   printf("Les valeurs finales sont  %d %d\n",var1,var2);
[20] }
[21]
[22] prog(prog_var1,prog_var2)  /* prog_var1 est une valeur entière */
[23] int prog_var1,*prog_var2;  /* prog_var2 pointe sur un type entier */
[24] {
[25]   printf("\nLes valeurs initiales dans prog sont %d %d\n",
[26]            prog_var1,*prog_var2);
[27]   prog_var1  = 135;
[28]   *prog_var2 = 172;
[29]   printf("Les valeurs modifiées dans prog sont %d %d\n\n",
[30]          prog_var1,*prog_var2);
[31] }
```

Dans ce programme sont déclarées deux variables entières *var1* et *var2*, qui sont bien des variables, et non des pointeurs. Des valeurs sont attribuées aux deux variables. Ensuite, dans la ligne 16 est activée la fonction *prog()*, à laquelle sont transmises en guise d'arguments la valeur de la variable *var1* d'une part, et l'adresse de la variable *var2* d'autre part. La définition de la fonction proprement dite est répartie sur les lignes 22 à 31, et vous voyez qu'elle s'attend à recevoir en guise de premier argument une variable entière, et en guise de second argument un pointeur dirigé sur des valeurs entières, c'est-à-dire l'adresse d'une variable entière. Les valeurs initialement transmises sont donc correctes, puisqu'elles correspondent exactement aux types des valeurs que la fonction s'attend à recevoir. L'adresse de *var2* sera traitée comme adresse, et non comme valeur de variable.

Lancez le programme.

Voici l'affichage retourné par le programme DEUXVOIE.C :

```
Les valeurs initiales de var1 et var2 sont 100 101
Les valeurs initiales dans prog       sont 100 101
Les valeurs modifiées dans prog       sont 135 172
Les valeurs finales                   sont 100 172
```

Vous constatez que la valeur de *var1* a bien été changée dans la fonction *prog()*, mais pas dans le programme principal. Par contre, la valeur de *var2* a également changé dans le programme principal après l'exécution de la fonction. Vous en connaissez la raison. Pour ce qui concerne var1, c'est une copie de sa valeur qui a été transmise à la fonction, dans laquelle elle a été traitée sous le nom *prog_var1*. Par conséquent, la fonction n'exerce aucune influence sur la valeur originale de *var1*, qui reste dans le programme principal. Pour *var2*, les choses se passent différemment : c'est son adresse qui a été transmise. Dès que la valeur de la variable qui se trouve à cette adresse est modifiée par la fonction, ce changement reste valable dès que vous revenez au programme principal.

Par conséquent, si nous utilisons des pointeurs dans une fonction appelée, nous avons la possibilité d'accéder aux données originales de notre choix, et de les modifier dans cette fonction.

10.7. Initialiser des tableaux de pointeurs (pointerarrays)

Pour finir ce chapitre, nous allons approfondir la technique d'utilisation de pointeurs, ainsi que le travail avec des chaînes et des fonctions, grâce au programme de démonstration POINTINI.C. Lorsque vous aurez parfaitement compris le sens de ce programme, vous pourrez déjà prétendre en savoir beaucoup sur le langage de programmation C.

NOM(0) = invalide
*NOM(0) = i

Listing 39 : Le programme POINTINI.C

```
[1 ] /* Chapitre 10.7 */
[2 ]
[3 ] /* Nom du programme : POINTINI.C */
[4 ] /* Exemple d'initialisation d'un tableau de pointeurs */
[5 ]
[6 ] char *jour_semaine(int nr);   /* Prototype - La fonction retourne */
[7 ]                 */ un pointeur sur char et attend un entier (Integer) */
[8 ]
[9 ]
[10] char *nom[] =
[11] {
[12]    0"invalide",
[13]    1"Dimanche",
[14]    2"Lundi",
[15]    3"Mardi",
[16]    4"Mercredi",
[17]    5"Jeudi",
[18]    6"Vendredi",
[19]    7"Samedi"
[20] };     /* Le tableau nom est initialisé. */
[21]        /* Il se compose de 8 pointeurs dirigés sur les chaînes */
[22]        /* présentées ici */
[23]
[24] main()
[25] {
[26]    int nombre;
[27]    printf("Veuillez entrer la valeur correspondant à
                un jour de semaine quelconque ");
[28]    scanf("%d",&nombre);
[29]    printf( "\n\nLe jour %d de la semaine est %s
                ",nombre,jour_semaine(nombre));
[30] }
[31]
[32] char *jour_semaine(nr)
[33] {
[34]    return( (nr < 1 || nr > 7) ? nom[0] : nom[nr] );
[35] }
```

Vous voyez, dans un premier temps, en ligne 6, le prototype de la fonction *jour_semaine()*. Vous constaterez qu'une fonction peut aussi retourner un pointeur en guise de réponse. Dans notre cas, il s'agit d'un pointeur dirigé sur des données de type *char*. En guise d'argument, c'est une valeur entière qui est transmise à cette fonction, que celle-ci traite sous le nom *nr*.

La déclaration qui suit, dans les lignes 10 à 20, montre comment il faut initialiser un tableau de pointeurs dirigés sur des données de type *char*. Une spécification de longueur est sans objet, puisque celle-ci est déterminée automatiquement par le compilateur en fonction du nombre d'éléments attribués. L'initialisation est réalisée par l'intermédiaire de l'opérateur d'affectation. Les éléments attribués

sont délimités par des accolades. Chaque élément pris individuellement est séparé de l'élément suivant par une virgule.

Dans le présent cas, ce sont des constantes alphanumériques qui sont attribuées. Il en résulte huit pointeurs dirigés chacun sur l'origine de la constante alphanumérique initialisée. Ainsi, *nom[0]* pointe sur la constante "*invalide*", et *nom[7]* sur *Samedi*. Vous avez donc à faire à des pointeurs dirigés sur des pointeurs. Le tableau *nom[]* est déclaré avant la fonction, et peut par conséquent être utilisé dans toute fonction de notre programme. Dans la logique du langage C, on parle d'un tableau global.

La déclaration est suivie de la programmation de *main()*. Par l'intermédiaire de la fonction de bibliothèque *scanf()*, un nombre saisi au clavier est affecté à la variable *nombre*. Cette fonction *scanf()* fera l'objet d'une étude détaillée dans la section 11.2. Pour l'instant, il suffit de savoir qu'avant l'instruction *printf*, un nombre a été chargé dans la variable *nombre*. L'instruction elle-même montre une fois de plus la possibilité d'utiliser dans une fonction (ici *printf()*) une autre fonction (ici *jour_semaine()*). Le résultat de la fonction *jour_semaine()* est introduit sous forme de chaîne alphanumérique dans la chaîne de mise en forme de la fonction *printf()*, à l'aide du descripteur de champ *%s*.

Voyons maintenant d'un peu plus près la fonction *jour_semaine()* dont la définition est répartie sur les lignes 32 à 35. La tâche de cette fonction consiste à déterminer le jour de la semaine qui correspond au nombre entier saisi au clavier. Une unique instruction contenant *return()* suffit. Celle-ci renvoie le pointeur dirigé sur *nom[0]* que le nombre saisi soit inférieur à 1 ou supérieur à 7. Dans tous les autres cas, c'est-à-dire lorsque le numéro du jour de la semaine saisi est correct, le pointeur correspondant sera renvoyé. Dans notre syntaxe, nous utilisons ici le *if* compact et l'opérateur | | qui symbolise le "ou" logique. Il en résulte un code de programmation très compact. Même si cette concision est obtenue au détriment de la lisibilité du programme, il n'en demeure pas moins qu'il s'agit là d'une syntaxe typique pour un programmeur C.

Lancez plusieurs fois le programme. Entrez les numéros de jour de votre choix. Vous constatez que le résultat renvoyé par le programme est toujours correct. Au fur et à mesure que vous gagnerez en expérience avec les pointeurs, vous découvrirez toutes les possibilités qui s'ouvrent à vous grâce à l'utilisation de ces outils. Si vous désirez rédiger des programmes efficaces et ambitieux, vous ne pourrez pas vous passer de pointeurs. C'est pourquoi nous vous conseillons de réétudier ce chapitre jusqu'à ce que vous soyez bien imprégnés de l'esprit lié au concept des pointeurs.

10.8. Récapitulation

Les pointeurs sont des éléments dirigés sur des variables. Ils contiennent l'adresse d'une variable, mais pas la valeur d'une variable. Les pointeurs doivent être déclarés pour chaque type de données. Le symbole de déclaration est l'astérisque (*). Lorsqu'un astérisque est écrit devant un nom de variable, c'est que cette variable est un pointeur dirigé sur une autre variable de même type. L'opérateur & détermine l'adresse d'une variable qui peut ensuite être affectée à un pointeur. Il n'existe aucune limite au nombre de pointeurs pouvant être dirigés sur une même variable.

L'instruction

```
*pt1 = 13;
```

est équivalente à

```
index = 13;
```

lorsque *pt1* est un pointeur dirigé sur la variable *index*. Il existe donc deux techniques permettant d'accéder à une variable : l'accès peut être réalisé directement par l'intermédiaire du nom de la variable, mais il peut aussi être réalisé au moyen d'un pointeur dirigé sur cette variable. Ces deux techniques produisent des résultats parfaitement identiques.

Arithmétique de pointeurs signifie que l'adresse contenue dans un pointeur peut être augmentée ou diminuée. Les opérations tolérées en arithmétique des pointeurs sont l'addition et la soustraction. Dans le cas de l'addition, l'accès sera repoussé vers l'adresse d'une cellule de mémoire qui suit. Pour ce qui concerne la soustraction, l'accès concernera une adresse d'une cellule précédente. Le nombre d'octets qui régit le pas des variations de la valeur d'un pointeur dépend du type de données pour lequel le pointeur avait été déclaré.

Grâce aux pointeurs il est possible de modifier toutes les variables que l'on désire, en-dehors d'une fonction. Dans ce but, il suffit de transmettre à la fonction les adresses des variables à modifier.

10.9. Un exemple pour bien comprendre

Avant de passer au chapitre suivant, nous vous suggérons d'analyser le programme CALENDR.C qui suit.

Listing 40 : Le programme CALENDR.C

```
[1 ] /* Chapitre 10.9 */
[2 ]
[3 ] /*Nom du programme : CALENDR.C*/
[4 ] /*Initialisation d'un tableau à deux dimensions */
[5 ] /*Exemple pour l'approfondissement du travail avec les pointeurs et */
[6 ] /*les tableaux, et pour l'utilisation d'opérateurs logiques */
[7 ]
[8 ] /*Idée : K & R, The C programming language*/
[9 ] /*Englewood Cliffs 1977*/
[10]
[11] /*****************************************************************/
[12] int table_des_jours[2] [13] =
[13] {
[14]   {0,31,28,31,30,31,30,31,31,30,31,30,31},/* Pas d'année bissextile*/
[15]   {0,31,29,31,30,31,30,31,31,30,31,30,31} /* Année bissextile */
[16] };
[17]
[18] char *nom[] =
[19]   {
[20]       "invalide",
[21]       "Janvier",
[22]       "Février",
[23]       "Mars",
[24]       "Avril",
[25]       "Mai",
[26]       "Juin",
[27]       "Juillet",
[28]       "Août",
[29]       "Septembre",
[30]       "Octobre",
[31]       "Novembre",
[32]       "Décembre"
[33]   };    /* Tableau avec pointeur sur les noms des mois */
[34]
[35] int annee, jours;
[36]
[37] char *mois(int nr);  /* Prototype - La fonction retourne un          */
[38]                      /* pointeur sur char et attend un nombre entier */
[39]
[40] int annee_bissextile(int annee);/* Prototype - Détection d'une
                                        année bissextile*/
[41]
[42] void jour_du_mois(int annee, int jour_dans_annee);
[43] /* Prototype - Calcul du jour dans le mois à partir du jour
        dans l'année */
```

```
[44]
[45] main()
[46] {
[47]   printf("Veuillez entrer un nombre désignant une année ");
[48]   scanf("%d",&annee);
[49]   printf("Veuillez entrer le nombre des jours écoulés ");
[50]   scanf("%d",&jours);
[51]   printf("\n\n");
[52]   jour_du_mois(annee,jours);
[53] }
[54]
[55]
[56]
[57]
[58]
[59] int annee_bissextile(annee)
[60] {
[61]   int i;
[62]
[63]   return((i=annee%4 == 0 && (annee%100 != 0 || annee%400==0)));
[64]   /* renvoie un faux logique = 0 si pas d'année bissextile, sinon */
[65]   /* un nombre positif et est alors vrai */
[66]
[67] }
[68]
[69] char *mois(nr)
[70] {
[71]   return( (nr < 1 || nr > 12) ? nom[0] : nom[nr] );
[72] }
[73]
[74] void jour_du_mois(int annee, int jour_dans_annee)
[75] {
[76]   int annee_b, i, m, j;
[77]
[78]   annee_b = annee_bissextile(annee); /* 0, pas d'année bissextile,
                                             sinon année bissextile */
[79]   for (i = 1; jour_dans_annee > table_des_jours[annee_b] [i]; i++)
[80]       jour_dans_annee -= table_des_jours[annee_b] [i];
[81]   m = i;                    /* Numéro du mois */
[82]   j = jour_dans_annee;      /* Jour du mois */
[83]   printf("L'année %d est %s",annee,
[84]   ((annee_b) ? "une année bissextile":"pas une année bissextile"));
[85]   printf("\nLe %dième jour de %d est le %dième jour du mois de %s",
[86]           jours,annee,j,mois(m));
[87] }
```

Nous ne mentionnerons aucun commentaire à propos de ce programme. Si, pour l'instant, vous n'arrivez pas encore complètement à l'assimiler, vous pourrez toujours vous y reporter au fur et à mesure que vous progresserez dans l'étude du langage C. Vous parviendrez ainsi à approfondir progressivement la fonction de ce programme.

Exercices se rapportant au chapitre 10

Exercice 1 :
Déclarez un tableau de caractères pouvant contenir un maximum de 80 caractères. A l'aide de la fonction *strcpy()*, copiez la chaîne

```
"chaîne_de_caractères"
```

dans ce tableau, puis ordonnez l'affichage de cette chaîne alphanumérique à l'aide d'un pointeur dirigé sur le tableau de caractères. Vous programmerez à cet effet une fonction

```
int output(char *string)
```

qui retourne, en guise de résultat, la longueur de la chaîne à afficher.

Exercice 2 :
Ecrivez un programme qui, à l'aide d'une fonction

```
int copier_string(char *s1, char *s2)
```

que vous programmerez, copie la chaîne *s1* dans la chaîne *s2*. Le résultat de la fonction devra être la longueur de la chaîne copiée.

Exercice 3 :
Ecrivez un programme qui, à l'aide d'une fonction

```
int test_string(char *s1, char *s2, int precision)
```

que vous programmerez, compare les chaînes *s1* et *s2*. L'argument "*precision*" devra permettre de définir le nombre de caractères à comparer. Si les chaînes de caractères composées du nombre de caractères défini par cet argument sont identiques, le programme doit renvoyer le nombre 1. Dans le cas contraire, il doit retourner 0.

Testez le programme avec les chaînes de caractères

```
pari
parisienne
```

et utilisez pour l'argument "*precision*" les valeurs 4 et 5.

11 Entrées/Sorties standard

Dans ce chapitre, vous apprendrez à écrire des routines d'entrée/sortie bien pratiques. Vous découvrirez les fonctions standard du C qui assurent la lecture des saisies effectuées au clavier. D'autres sujets traitent du chargement de divers types de données, de la saisie avec et sans écho, et de l'entrée/sortie dans la mémoire de travail. Par la même occasion, nous développerons notre propre routine, par l'intermédiaire de laquelle des chaînes alphanumériques de longueur quelconque pourront être chargées dans un tableau.

Les opérations d'entrée/sortie standard se rapportent au périphérique d'entrée standard, le clavier, et au périphérique standard de sortie, l'écran. En MS-DOS, ces deux appareils sont désignés par le terme "console", et leur désignation normalisée est CON. Etant donné que ces périphériques sont fréquemment utilisés, il n'est pas nécessaire de les spécifier explicitement dans la programmation. Le système considère que, par défaut, les saisies sont effectuées au clavier, et les sorties doivent être dirigées sur l'écran. Dans les rubriques qui suivent, nous ferons nos premiers pas dans la programmation des entrées/sorties en C. Le programme ci-après charge les caractères entrés au clavier jusqu'à ce que soit actionnée la combinaison de touches <Shift>+<x>.

Listing 41 : Le programme IOSIMPLE.C

```
[1 ] /* Chapitre 11 */
[2 ]
[3 ] /* Nom du programme : IOSIMPLE.C */
[4 ] /* Chargement et renvoi de caractères - Mémorisation temporaire des */
[5 ] /* entrées assurée par le buffer du DOS */
[6 ] #include "stdio.h"      /* En-tête standard pour Entrées/Sorties */
[7 ]
[8 ] main()
[9 ] {
[10]   char c;
[11]   printf("Entrez une lettre - X = Fin.\n");
[12]   do
[13]   {
[14]      c = getchar();      /* Lecture d'un caractère saisi au clavier */
[15]      putchar(c);         /* Renvoi du caractère à l'écran */
[16]   } while (c != 'X');    /* jusqu'à pression sur X */
[17]
[18]   printf("\nFin du programme.\n");
[19] }
```

Dans un premier temps, en ligne 6, vous intégrez dans le code du programme le fichier d'en-tête STDIO.H à l'aide de

```
#include "stdio.h";
```

Ce fichier renferme des constantes symboliques et des macros en guise de définitions implicites pour les opérations d'entrée/sortie. Ce fichier est appelé "Fichier d'en-tête" parce qu'on l'intègre au début du texte du programme, dans son en-tête, les guillemets font en sorte que le compilateur recherche d'abord le fichier d'en-tête dans le répertoire courant, puis dans celui de l'environnement du compilateur.

Vous pouvez constater que, avec votre compilateur C, sont livrés un certain nombre de fichiers d'en-tête, possédant tous l'extension ".H", et couvrant chacun un domaine spécifique de programmation. Généralement, les fichiers d'en-tête sont enregistrés dans un répertoire appelé \INCLUDE, et peuvent être intégrés à tout programme.

En langage C, contrairement à d'autres langages de programmation, aucune opération d'entrée/sortie n'est définie comme partie du langage. Ces opérations doivent être programmées par l'utilisateur. Toutefois, la rédaction systématique de routines d'entrée/sortie n'intéressant personne, les programmeurs de Quick C, Turbo C et d'autres compilateurs C ont pris ce travail à leur charge, en développant des constantes, macros et fonctions appropriées à ces routines. Ces éléments sont à présent livrés avec le compilateur.

Imprimez le fichier STDIO.H, et étudiez son contenu. De nombreuses notions vous sembleront encore incompréhensibles. Par contre, certains éléments de rédaction ne vous paraîtront pas étranger. Vous trouvez ici des constantes et des macros définies par *#define*. Ne vous inquiétez pas si le contenu de ce fichier vous est globalement incompréhensible. Dans un premier temps, notre unique intention est d'exploiter ce fichier.

11.1. Lire des caractères

Venons-en à notre programme de démonstration. Nous y voyons la déclaration d'une variable *c* de type *char*, puis une instruction d'affichage d'une invitation à la saisie à l'aide de *printf()*. Ensuite, en ligne 12, commence une boucle *do..while* qui ne s'achèvera que lorsque *c* recevra la valeur *X*. Cette condition d'exécution de boucle (c doit être différent du caractère *X*) figure dans la ligne 16. Vous avez par conséquent à faire à une boucle pilotée par la fin, qui sera parcourue au minimum une fois.

Intéressons-nous maintenant aux deux fonctions contenues dans la boucle qui, sont nouvelles. La première fonction, *getchar()* lit un caractère provenant du périphérique d'entrée standard, c'est-à-dire du clavier. Ce caractère est attribué à la variable *c* par l'intermédiaire de l'instruction

```
c = getchar();
```

La deuxième fonction, *putchar()*, utilise le périphérique standard de sortie, l'écran, et y renvoie le contenu de la variable *c*. Le caractère est ainsi toujours affiché à la position occupée par le curseur, puis ce dernier est décalé d'un cran vers la droite. Par conséquent, cette boucle de notre programme lira des données entrées au clavier, et les affichera à l'écran, jusqu'à ce que l'usager appuie sur <Shift>+<x>.

Lancez le programme. Vous serez étonné de constater que, dans l'immédiat, les caractères que vous entrez sont renvoyés à l'écran, même lorsque vous aurez appuyé sur <X> et que vous continuerez de saisir d'autres caractères. Par contre, dès que vous appuyez sur la touche <Return> pour conclure votre saisie, la ligne validée par <Return> sera une nouvelle fois affichée sur l'écran. Mais cette fois, jusqu'à notre caractère de fin de saisie, le <X>

Affichage renvoyé pendant la saisie :

```
abcdsXjhfhkjlk
```

Affichage renvoyé après pression sur <Return> :

```
abcdsX
```

Dans la section qui suit, nous verrons pourquoi l'affichage a pris cette forme.

11.1.1. Saisie par l'intermédiaire du buffer de MS-DOS

Dans ce chapitre, nous aborderons des techniques fondamentales de travail dans l'environnement du système d'exploitation MS-DOS. Lorsque, sous contrôle de MS-DOS, des données sont chargées depuis le clavier, le système d'exploitation les mémorise dans une mémoire intermédiaire appelée buffer (ou mémoire tampon), jusqu'à ce que l'utilisateur appuie sur la touche <Return>. Ensuite, le contenu du buffer est transmis sous forme de chaîne au programme qui fonctionne sous MS-DOS. Toutefois, chaque caractère est déjà affiché à l'écran pendant la saisie. Ce processus est appelé Echo.

L'explication du mode de fonctionnement de IOSIMPLE.C est maintenant évidente. Vous voyez d'abord l'écho de votre saisie clavier, puis, lorsque vous appuyez sur <Return>, vous revoyez les mêmes saisies. Mais dans cette seconde phase, les caractères ont été communiqués à notre programme par MS-DOS sous forme de chaîne, et conformément à l'instruction contenue dans la boucle, en ligne 15, elles ont été affichées une seconde fois.

Lancez une nouvelle fois notre programme, et entrez des caractères quelconques, dont quelques <X> vers le milieu de la chaîne saisie, puis d'autres lettres de votre choix. Appuyez ensuite sur <Return>. Vous constatez que l'écho de MS-DOS renvoie des caractères à l'écran jusqu'à ce que vous appuyez sur <Return>. Le caractère <X> n'a aucune incidence sur l'écho, ni sur le buffer d'entrée. Ce n'est qu'après pression sur <Return> que les caractères sont transmis à notre programme, qui, conformément aux conditions qu'il renferme, arrête l'affichage à la suite du premier <X>.

Dans la section qui suit, nous allons voir comment l'on peut contourner le buffer d'entrée.

11.1.2. Saisie de données sans passer par le buffer

Le programme ci-après montre le principe de la saisie de données ne faisant pas appel au buffer du DOS. Vous constaterez que la saisie sera stoppée immédiatement après une pression sur <X>.

Listing 42 : Le programme NOECHO.C

```
[1 ] /* Chapitre 11.1.2 */
[2 ]
[3 ] /* Nom du programme : NOECHO.C */
[4 ] /* Chargement caractère par caractère avec sortie sans passer */
[5 ] /* par un buffer */
[6 ] #include "stdio.h" /* En-tête standard pour Entrées/Sorties */
[7 ]
[8 ] main()
[9 ] {
[10]   char c;
[11]
[12]   printf("Entrez une lettre , X = Fin.\n");
[13]
[14]   do
[15]   {
[16]       c = getch();          /* Caractère saisi au clavier */
[17]       putchar(c);           /* Renvoi du caractère sur l'écran */
[18]   } while (c != 'X');       /* jusqu'à pression sur X */
[19]
[20]   printf("\nFin du programme.\n");
[21] }
```

La différence par rapport au premier programme réside dans l'utilisation de la fonction *getch()*, en ligne 16. Celle-ci charge directement dans le programme la saisie effectuée au clavier, en contournant l'écho. La pression sur <Return> devient ainsi inutile, et la condition d'interruption de saisie est immédiatement reconnue.

Mais comment faut-il faire pour écrire un programme qui interprète une action sur <Return> comme saut de ligne, et dans lequel l'on puisse malgré tout utiliser la fonction *getch()* ? Chargez à cet effet le programme RETURN.C.

11.1.3. Saisie avec saut de ligne programmé

En principe, vous avez l'habitude qu'une action sur la touche <Return> positionne le curseur dans une nouvelle ligne sur l'écran, au début de cette ligne. Ce saut de ligne n'est pas automatique. Il faut le programmer.

Listing 43 : Le programme RETURN.C

```
[1 ] /* Chapitre 11.1.3 */
[2 ] /* Nom du programme : RETURN.C */
[3 ] /* Programme de saisie amélioré, sans buffer d'entrée  */
[4 ]
[5 ] #include "stdio.h"
[6 ] #define CR 13  /* Le code ASCII de Carriage Return est 13 */
[7 ] #define LF 10  /* Le code ASCII de LINE FEED est 10       */
[8 ]
[9 ] main()
[10] {
[11]   char c;
[12]
[13]   printf("Entrez un caractère, X = Fin.\n");
[14]
[15]   do
[16]   {
[17]     c = getch();  /* Chargement d'un caractère */
[18]     if (c != 'X') putchar(c);/* Montre la touche, si différente de X */
[19]     if (c == CR) putchar(LF);/* Si pression sur RETURN, */
[20]                              /* alors saut de ligne */
[21]   } while (c != 'X');
[21]
[22]   printf("\nFin du programme.\n");
[23] }
```

Dans ce but, nous déclarons dans notre programme de démonstration les constantes *LF* et *CR*, dans les lignes 6 et 7. *LF* signifie Line Feed, expression anglaise qui désigne le saut de ligne, et *CR* signifie Carriage Return, l'expression anglaise qui désigne un retour chariot. Dans la boucle qui suit, chaque saisie effectuée au clavier est vérifiée, pour voir s'il ne s'agit pas du code ASCII 13 (ligne 19) qui signale à l'ordinateur une action sur la touche <Return>. Si tel est le cas, l'instruction

```
putchar(LF);
```

qui suit positionnera le curseur dans la ligne suivante. Ce mouvement est déclenché par le renvoi de la constante symbolique *LF*, dont le code ASCII est 10.

Nous pourrions aussi nous dispenser des deux constantes symboliques, et écrire dans la boucle l'instruction

```
if (c == 13) putchar(10);
```

La méthode que nous utilisons contribue toutefois à faciliter la compréhension du code du programme, et est par conséquent le style de programmation le mieux adapté.

Lancez le programme. Vous voyez que l'affichage se conforme aux règles énoncées. La touche <Return> provoque un retour chariot suivi d'un saut de ligne, et l'exécution du programme est interrompue immédiatement après la saisie de <X>.

Vous venez de découvrir deux méthodes qui permettent de charger des variables depuis le clavier. Revoyons rapidement chacune d'entre elles.

Lorsque vous utilisez la première méthode, et que vous chargez les données par l'intermédiaire de la fonction *getchar()*, vous avez la possibilité de mémoriser les saisies effectuées au clavier, et de ne les transmettre au programme qu'après validation par <Return>. Cette méthode ne permet toutefois pas de communiquer directement au programme une touche unique. Il est indispensable de valider la saisie par <Return>, mais en contrepartie vous avez la possibilité de corriger d'éventuelles fautes de frappe.

Cette possibilité n'existe pas si les données sont chargées directement dans le programme à l'aide de *getch()*. Le caractère saisi est immédiatement à notre disposition dans ce cas. Le programme attend une action sur une touche, puis passe immédiatement à l'instruction suivante. Cette méthode présente l'avantage, lorsque vous écrivez des programmes interactifs, de faire dépendre étroitement l'exécution de celui-ci des saisies de l'utilisateur.

Jusqu'ici, nous n'avons chargé que des caractères dans des variables de type *char*. Dans la section qui suit, vous apprendrez comment entrer des nombres, et quelles sont les sources d'erreurs dont il faut se méfier.

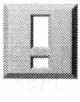
11.2. Lecture de valeurs entières

Le programme LIRINT.C est un exemple de lecture structurée de valeurs entières. Il s'achève lorsque l'on saisit le nombre 100. La structure de ce programme équivaut à celle des exemples précédents.

Listing 44 : Le programme LIRINT.C

```
[1 ] /* Chapitre 11.2 */
[2 ]
[3 ] /* Nom du programme : LIRINT.C */
[4 ] /* Saisie de valeurs entières (type int) */
[5 ]
[6 ] #include "stdio.h"
[7 ]
[8 ] main()
[9 ] {
[10]   int valeurin;
[11]
[12]   printf("Entrer un nombre compris entre 0 et 32767 - 100 = Fin \n");
[13]
[14]   do
[15]   {
[16]       scanf("%d",&valeurin); /* Lecture d'une variable entière */
[17]       if (valeurin != 100)
[18]           printf("La valeur est %d\n",valeurin);
[19]       if (valeurin == 100)
[20]           printf("En entrant 100, vous avez quitté le programme\n");
[21]
[22]   } while (valeurin != 100);
[23]
[24]   printf("Fin du programme\n");
[25] }
```

La différence par rapport aux exemples précédents réside dans le fait que cette fois, nous chargeons la variable entière *valeurin* dans la boucle *do..while* jusqu'à ce que soit saisi le nombre 100. Nous utilisons à cet effet la fonction *scanf()*, très semblable à la fonction *printf()* que nous connaissons déjà. Dans cette fonction, il nous faut aussi spécifier le symbole de conversion pour les valeurs saisies. Dans notre cas, il s'agit de *%d* qui signale que la structure de la saisie est de forme décimale.



Il faut savoir que la fonction *scanf()* ne s'attend pas à recevoir la variable *valeurin* en guise d'argument, mais l'adresse de cette variable, c'est-à-dire *&valeurin*.

scanf() est une fonction qui doit toujours être appelée par une autre fonction, à laquelle elle retournera des valeurs. C'est justement dans ce but que les fonctions utilisent l'adresse d'une variable (voyez à ce sujet la section 10.7). La fonction *scanf()* lit les données dans le buffer d'entrée jusqu'à ce que la saisie soit conclue par <Return>. A cette occasion, les espaces figurant en tête de saisie ne sont pas pris en compte. Chaque valeur lue sera mémorisée dans la variable spécifiée à l'intérieur des parenthèses de la fonction.

L'instruction

```
scanf("%d",&valeur);
```

signifie par conséquent :

"lis une valeur entière dans le buffer, et mémorise-la à l'adresse à laquelle se trouve la variable *valeur*."

Dans notre programme, cette instruction sera répétée jusqu'à ce que la valeur 100 soit entrée au clavier.

Lancez le programme, et entrez plusieurs nombres parmi lesquels il y aura aussi le nombre 100. Séparez les diverses saisies par un espace, puis concluez l'entrée par une pression sur <Return>. Vous constatez que, dans notre programme, toutes les données sont d'abord mémorisées dans le buffer du système d'exploitation, et que la transmission au programme ne s'effectue qu'à la suite d'une pression sur la touche <Return>. Le programme proprement dit renvoie les valeurs séparées par des espaces jusqu'à ce qu'il rencontre le nombre 100. Les nombres éventuellement saisis à la suite de 100 n'apparaissent plus à l'écran, puisque dans la ligne 22 la valeur 100 conditionne l'interruption de la boucle de renvoi :

```
while (valeurin != 100);
```

11.3. Erreurs avec scanf()

Lancez une nouvelle fois le programme, et entrez les valeurs suivantes :

```
2000 32768 65536 100
```

Sur votre écran sont toutefois renvoyés les nombres :

```
2000
-32768
0
```

Il ne s'agit manifestement là pas des valeurs que vous aviez saisies. Vous n'avez fait aucune erreur : la cause de cet affichage erroné est à rechercher dans le principe suivant lequel les variables entières sont mémorisées en C. Comme vous le savez, la capacité mémoire réservée pour des variables entières est de deux octets soit seize bits. De ces seize bits, le plus significatif est le seizième, qui est utilisé pour le signe de la variable. En définitive, il ne reste donc que quinze bits pour la valeur absolue de cette variable qui pourra par conséquent être comprise entre -32768 et 32767. Toute valeur supérieure ou inférieure à ces bornes ne peut donc en aucun cas être retournée correctement.

Vous venez de voir un nouvel exemple qui prouve que la responsabilité de la déclaration et de l'affectation d'une valeur correctes à des variables relèvent exclusivement du programmeur.

Lancez une fois encore notre programme, et variez les saisies. Entrez maintenant des valeurs qui n'appartiennent pas à l'intervalle de définition des variables entières, puis vérifiez l'affichage produit. Essayez de découvrir le principe selon lequel sont traitées les valeurs extérieures à ce domaine de définition. Pour finir, nous vous suggérons d'entrer également quelques caractères non valables (des lettres, par exemple) et d'observer les réactions du système.

Vous constaterez que, lors de la saisie de caractères imprévus, un défaut du programme sera mis en évidence : celui-ci entame un cycle d'exécution infini. La cause de ce comportement est à rechercher dans la fonction *scanf()*, qui n'est pas capable d'intercepter des saisies erronées. Dans la section 11.5, vous écrirez des routines de saisie qui intercepteront des erreurs de saisie.

11.4. Lecture de chaînes

Après nous être consacrés à la saisie de valeurs entières, nous allons étudier le principe s'appliquant à la saisie de chaînes de caractères. Le programme qui suit lit des chaînes entrées au clavier jusqu'à ce que soit saisi un mot commençant par la lettre X.

Listing 45 : Le programme STRINGIN.C

```
[1 ] /* Chapitre 11.4 */
[2 ]
[3 ] /* Nom du programme : STRINGIN.C */
[4 ] /* Exemple de saisie de variables alphanumériques */
[5 ]
[6 ] #include "stdio.h"
[7 ]
[8 ] main()
[9 ] {
[10]   char string[25];
[11]
[12]   printf("Entrez un maximum de 25 caractère.\n");
[13]   printf("La lettre X dans la première colonne
            achève le programme.\n");
[14]
[15]   do
[16]   {
[17]       scanf("%s",string);
[18]       printf("L'ordre chronologique de la saisie est
                \t ->\t %s\n",string);
[19]   } while (string[0] != 'X');
[20]
[21]   printf("Fin du programme.\n");
[22] }
```

Deux différences par rapport au programme LIRINT.C sont évidentes. Il y a d'une part, en ligne 10, la définition d'une chaîne, et d'autre part, dans la ligne 17, une instruction contenant *scanf()*, dans laquelle est utilisé le descripteur de champ *%s* qui s'applique au chargement d'une chaîne. Vous n'avez pas besoin d'utiliser l'opérateur d'adressage & pour transmettre à *scanf()* l'adresse de la variable de type *chaîne*. La variable *string* est un tableau et ainsi, par définition une adresse. La logique de notre exemple de programme n'est pas nouvelle. Aucune explication supplémentaire n'est donc nécessaire sur ce sujet.

Lancez le programme, et entrez la phrase suivante que vous conclurez par <Return>.

Saisie destinée à STRINGIN :

```
Ceci est une phrase.
```

L'affichage vous surprendra probablement, car la phrase est renvoyée mot par mot.

Affichage retourné par STRINGIN :

```
L'ordre chronologique de la saisie est
Ceci est une phrase.
```

L'explication de ce comportement est une fois de plus à rechercher dans le mode de fonctionnement de la fonction utilisée. La fonction *scanf()* lit les caractères d'une chaîne jusqu'à ce qu'elle rencontre un espace ou la fin du buffer d'entrée. C'est pourquoi chaque mot est chargé séparément dans la boucle, et immédiatement affiché. Nous avons programmé la boucle qui commence en ligne 16 de manière à ce qu'elle s'arrête lorsque le premier caractère d'une chaîne est <X>. Par conséquent, chaque mot dont l'initiale est <X> provoquera une interruption de fonctionnement de la boucle. Testez le programme, et entrez cinq mots séparés par des espaces, le troisième ayant <X> comme initiale. Vous constaterez que les trois premiers mots sont renvoyés correctement, et que les deux derniers ne sont pas retournés du tout.

Lors de la lecture de chaînes, la fonction *scanf()* souffre d'un grand inconvénient. Il n'est pas possible de contrôler efficacement les saisies de l'utilisateur. Cela signifie que, si l'utilisateur entre trop de caractères, vous courez le risque d'un plantage du programme. Celui-ci est lié au fait que le nombre des caractères saisis est supérieur au nombre des cellules de mémoire réservées pour une chaîne.

11.5. Lecture de chaînes de longueur quelconque

Dans les paragraphes qui suivent, nous allons vous présenter une routine de saisie qui permet de lire des chaînes de longueur quelconque, et qui d'autre part évite toute saisie excédentaire de caractères. Nous profiterons de cette occasion pour répéter les structures et techniques de programmation que nous avions vues jusqu'ici.

Listing 46 : Le programme INPUT.C

```
[1 ] /* Chapitre 11.5 */
[2 ]
[3 ] /* Nom du programme : INPUT.C */
[4 ] /* Exemple de routine de saisie avec correction d'erreurs */
[5 ]
[6 ] #include "stdio.h"
[7 ]
[8 ] void saisie(int l, char strg[]); /* pas de renvoi de valeur */
[9 ]
[10] #define BEEP    7   /* Signal sonore lorsque la fin de la chaîne
                           est atteinte */
[11] #define RETURN  13  /* code ASCII de RETURN */
[12] #define MAX     20  /* Longueur maximale d'une chaîne alphanumérique */
[13]
[14] main()
[15] {
[16]   char string[80];  /* soit exactement une ligne */
[17]
[18]  printf("Entrez un maximum de %d caractères, .\n",MAX);
[19]  saisie(MAX, string);
[20]  printf("\nVous avez entré : %s\n", string);
[21]  printf("Fin du programme.........");
[22] }
[23]
[24]
[25] void saisie(int l, char *strg) /* Chargement de l caractères */
[26] {
[27]   int i = 0;                   /* Compteur de caractères */
[28]    int a;                      /* nombre maximum de caractères */
[29]    char c;                     /* Mémoire tampon de caractères */
[30]
[31]    a = l - 1;          /* Longueur maximale de la chaîne de caractères */
[32]
[33]    while ( (c = getch()) != RETURN)
[34]    {
[35]      if (i < a)                /* Nombre maximum de caractères atteint ?*/
[36]      {
[37]        putchar(c);             /* Renvoi de caractère */
[38]        *(strg + i) = c;        /* Chargement du caractère dans la chaîne */
[39]        i++;                    /* caractère suivant */
[40]      }
[41]    else
[42]        putchar(BEEP);
[43]    }
[44]    *(strg+i) = NULL;           /* Fin de la chaîne alphanumérique */
[45]  }
```

Dans la ligne 8, vous voyez la définition d'un prototype, une fonction *saisie()* qui prévoit de recevoir comme arguments une variable entière et une chaîne de caractères. La longueur de la chaîne n'est pas limitée. Dans la suite du programme, nous définissons des constantes symboliques dans les lignes 10 à

12, *BEEP* (valeur ASCII 7) correspondant au renvoi d'un signal sonore par l'intermédiaire du haut-parleur intégré à votre machine.

La logique du programme principal n'est pas nouvelle. Vous pouvez donc vous concentrer sur la fonction *saisie()* des lignes 25 à 45. Dans les lignes 15 à 27 sont déclarées trois variables locales. En ligne 31, le programme détermine le nombre maximal des caractères autorisés dans la saisie, par l'intermédiaire des variables transmises, compte-tenu qu'une chaîne doit être conclue par '\0'.

Dans la boucle de saisie qui suit, vous voyez la routine d'entrée proprement dite. Celle-ci sera parcourue aussi longtemps que la touche actionnée détectée sans écho par la fonction *getch()* n'est pas <Return>. Le corps de la boucle est constitué d'une structure *if..else*, qui n'affiche les caractères entrés à l'écran tout en les chargeant dans la chaîne d'entrée qu'aussi longtemps que le nombre des caractères saisis est inférieur au maximum. Dès que le nombre maximal de caractère saisis est atteint, il n'y a plus que l'instruction de la ligne 42 qui est exécutée. Il en résulte des bips sonores retournés par l'ordinateur, à l'exclusion de toute autre action. La boucle d'entrée ne peut être achevée que par <Return>. Ce n'est qu'à la suite de cette action que, dans la ligne 44, le symbole NULL de fin de chaîne sera ajouté à l'extrémité de la chaîne.

Cet exemple de programme met une fois de plus en évidence qu'une chaîne peut être adressée soit comme *tableau* (array), soit par l'intermédiaire d'un pointeur dirigé sur le type de données *char*.

Testons le programme. Vous constaterez qu'il n'est pas possible d'entrer plus de caractères que le nombre prédéfini dans MAX. Dès que vous modifiez la valeur de MAX, vous voyez que la routine de saisie contrôle effectivement votre saisie. Les plantages de programme seront dorénavant évités.

11.6. Entrées/Sorties dans la mémoire de travail

Pour finir ce chapitre, voyons encore comment il est possible de lire et d'écrire directement dans la mémoire de travail, en C. Dans le programme qui suit, nous écrirons une chaîne dans la mémoire de travail, puis l'extrairons à nouveau de celle-ci par lecture. Par la même occasion, vous découvrirez une nouvelle fonction permettant de renvoyer des chaînes à l'écran.

Listing 47 : Le programme INMEM.C

```
[1 ] /* Chapitre 11.6*/
[2 ] /* Nom du programme : INMEM.C*/
[3 ] /* Entrées/Sorties en mémoire de travail - Adressage de chaîne */
[4 ]
[5 ] main()
[6 ] {
[7 ]   int nombres[5], resultats[5], index;
[8 ]   char ligne[80];
[9 ]
[10]   nombres[0] = 74;
[11]   nombres[1] = 18;
[12]   nombres[2] = 33;
[13]   nombres[3] = 30;
[14]   nombres[4] = 97;
[15]
[16]   sprintf(ligne,"%d %d %d %d %d\n",nombres[0],nombres[1],
[17]           nombres[2],nombres[3],nombres[4]);
[18]
[19]   printf("%s\n",ligne); printf("La ligne qui suit ne sera pas
                                structurée à l'affichage\n");
[20]   puts(ligne);
[21]   sscanf (ligne,"%d %d %d %d      %d",
[22]          &resultats[4],&resultats[3],
[23]          (resultats+2),(resultats+1),resultats);
[24]
[25]    for (index = 0;index < 5;index++)
[26]    printf("Le résultat final est %d\n",resultats[index]);
[27]
[28]  }
```

Dans ce programme sont d'abord déclarées quelques variables, puis, dans les lignes 10 à 14, est initialisé le tableau *nombres*. Les valeurs du tableau sont ensuite écrites dans la variable *ligne* de type chaîne, à l'aide de la fonction *sprintf()* (de l'anglais storage = printf en mémoire de travail). Par conséquent, ces valeurs se retrouvent directement dans la mémoire de travail. Par rapport à la fonction *printf()*, la différence réside dans le fait que le moyen de sortie n'est pas l'écran, mais une variable de type *chaîne* à spécifier en guise de premier argument. Ainsi, nous "écrivons" dans la mémoire de travail. Le "monde extérieur" ne voit rien. Toutefois, la chaîne *ligne* existe bien, et grâce à l'instruction qui suit (ligne 19)

```
printf("%s",ligne);
```

vous pouvez la rendre "visible". La fonction *sprintf()* utilise par conséquent comme "périphérique de sortie" la mémoire de travail, alors que *printf()* se rapporte à l'écran. L'instruction

```
puts(ligne);
```

qui figure dans la ligne 20 répète le renvoi de la chaîne écrite dans la mémoire de travail. La fonction utilisée, *puts()*, peut être utilisée de même que *printf()* pour retourner des chaînes de caractères. Toutefois, avec cette fonction, le renvoi est "non structuré". Cela signifie qu'à la différence de *printf()*, il n'est pas possible d'intégrer des variables et des séquences d'échappement à l'instruction. En contrepartie, *puts()* est plus compact, et par conséquent plus rapide.

Les lignes de programme qui suivent montrent qu'il est également possible de lire des valeurs dans la mémoire de travail, afin de les transmettre à des variables. La fonction *sscanf()* de la ligne 21 lit des caractères dans la mémoire de travail. A cet effet, il nous faut d'abord faire savoir à cette fonction dans quelle plage de la mémoire elle doit exercer son action de lecture. Suivent alors les descripteurs de champs puis les variables dans lesquelles sont transférées les valeurs lues. Il convient ici aussi de veiller, lors de la lecture de données qui ne sont pas des pointeurs, à utiliser les pointeurs dirigés sur les variables, et non les noms de variables eux-mêmes.

Les lignes 19 à 21 du programme montrent une nouvelle fois les diverses possibilités qui permettent de spécifier l'adresse d'une variable. La première technique utilisée est celle qui fait appel à l'opérateur d'adressage &. Ensuite est exploité le fait qu'un tableau est un pointeur, et que celui-ci peut être indexé.

Lancez le programme. Vous avez certainement remarqué que les variables de type *tableau* sont lues dans leur ordre inverse, et que l'affichage des éléments d'un tableau (ligne 24) commence par conséquent par la dernière variable.

11.7. Récapitulation

Le langage C permet de bénéficier d'une grande souplesse dans la programmation des entrées/sorties au clavier (périphérique de saisie implicite). Le programmeur peut choisir entre une saisie avec et sans écho. Il dispose pour la lecture de caractères des fonctions *getchar()* et *getch()*. Le renvoi est réalisé sur l'écran (périphérique implicite) par l'intermédiaire de *putchar()*. Les chaînes peuvent être affichées directement à l'aide de la fonction *puts()*.

L'avantage de la fonction *getchar()* réside dans le fait que la saisie peut encore être corrigée avant la validation par <Return>. Pour ce qui concerne la fonction *getch()*, cette technique n'est pas utilisable. Dans ce cas, chaque touche actionnée est immédiatement mémorisée.

La saisie structurée de nombres et de chaînes est programmée à l'aide de la fonction *scanf()*. La fonction s'attend à recevoir l'adresse de la variable dans laquelle les éléments lus devront être transcrits. La mise en forme est réalisée au moyen de descripteurs de champs. Les données que doit lire *scanf()* seront séparées par des espaces. Lorsque, lors de la lecture de chaînes, le nombre de caractères saisis est supérieur à la quantité prévue, il y a risque de blocage du programme. Cet inconvénient peut être contourné grâce à des routines de saisie écrites par le programmeur lui-même.

En C, il est possible de lire et d'écrire directement des données dans la mémoire de travail. L'écriture dans la mémoire de travail est programmée à l'aide de *sprintf()*. La lecture depuis la mémoire de travail est réalisée au moyen de *sscanf()*.

Exercices se rapportant au chapitre 11

Exercice 1 :
Ecrivez un programme qui charge des caractères entrés au clavier, en les affichant immédiatement sur l'écran. Un même caractère devra être simultanément renvoyé sous forme de caractère ASCII, de nombre décimal et de valeur hexadécimale. Utilisez le symbole du dollar comme critère d'interruption de la boucle de saisie.

Exercice 2 :
Ecrivez un programme qui, à l'aide d'une fonction

```
table_ascii()
```

que vous rédigerez vous même, renvoie à l'écran les caractères affichables du code ASCII. Vous ferez en sorte que, dans chaque ligne, soient affichés dix caractères au maximum.

Exercice 3 :
Ecrivez un programme qui, à l'aide de la fonction

```
char get_keyboard()
```

n'accepte de lire que les caractères 1 à 9 du clavier. Dès que l'utilisateur appuie sur une autre touche, le programme doit produire un signal sonore. La touche actionnée devra être renvoyée à la fonction appelante sous forme de valeur produite par la fonction appelée. Dans le programme principal, utilisez les lignes de programme suivantes :

```
main()
{
  printf("\nLe chiffre entré est %c",get_keyboard());
}
```

En annexe, vous trouverez deux propositions de solution pour cet exercice. Le programme EXO113 est une version utilisant la structure *switch*. Le programme EXO1131 utilise un code source plus rapide et plus compact, mais plus difficile à comprendre.

12 Gestion des fichiers en C

Dans ce chapitre, nous apprendrons tout ce qui concerne le traitement des fichiers en C. Vous ouvrirez, lirez, écrirez des fichiers, et apprendrez les fonctions standard nécessaires à partir de programmes de démonstration. Un autre sujet important concerne les diverses possibilités de lecture et d'écriture de caractères, de chaînes de caractères et de lignes. Vous trouverez ici un premier exemple de transmission d'arguments à l'en-tête *main()*. Le dernier sujet qui sera traité se rapporte à la sortie de fichiers sur l'imprimante.

Jusqu'ici, nous n'avons eu à faire qu'à des programmes dont les valeurs mémorisées dans des variables "disparaissent" à la fin du programme, car elles n'existent que dans la mémoire de travail. Dans ce chapitre, nous apprendrons à accéder à des données externes à la mémoire de travail, et à mémoriser durablement des variables (ou plutôt leur contenu) sur des mémoires de masse telles que le disque dur ou la disquette. Dans ce but, nous ferons appel à des fichiers.

La gestion de fichiers dans l'environnement du système d'exploitation (créer, copier, effacer etc..) relève du système d'exploitation lui-même, et ne constitue pas l'objet de notre étude. Ce qui nous intéresse, ce sont les possibilités de lire et de modifier le contenu de fichiers. Dans ce but, il nous faut d'abord clarifier quelques notions fondamentales.

Lorsque, dans la suite, nous parlerons de lecture (en anglais : read), nous supposerons qu'il s'agit d'un accès à des données. Lorsque nous modifierons des données, nous parlerons d'écriture (en anglais : write).

De plus, il est possible d'ajouter des données à la fin d'un fichier existant (en anglais : append).

12.1. Ecrire dans un fichier

Commençons par les règles fondamentales :

Avant de pouvoir travailler avec un fichier, il faut l'ouvrir.

Lorsque nous avons fini de travailler sur un fichier, il faut le refermer. Tout fichier que vous désirez écrire doit être créé. S'il existe un fichier du même nom, l'ancien contenu sera perdu.

Des données peuvent être ajoutées à un fichier existant (opération qui, en anglais est appelée *append*). Lorsque le fichier n'existe pas encore, il sera créé. Le contenu d'un fichier existant ne sera, dans ce cas, pas perdu.

Lorsque vous vous contentez de lire un fichier, son contenu ne subit aucune modification.

12.1.1. Ecrire des chaînes

Le programme de démonstration qui suit doit vous familiariser avec cette nouvelle technique de programmation. Ce programme, ECRIS.C, sera notre premier exemple d'écriture dans un fichier.

Listing 48 : Le programme ECRIS.C

```
[1 ] /* Chapitre 12.1.1 */
[2 ] /* Nom du programme : ECRIS.C */
[3 ] /* Exemple d'écriture dans un fichier  */
[4 ]
[5 ] #include <stdio.h>
[6 ]
[7 ] main()
[8 ] {
[9 ]   FILE *fp;        /* *fp est un pointeur sur un fichier (FilePointer) */
[10]   char string[25];
[11]   int index;
[12]
```

```
[13]   fp=fopen("LIGNES.TXT","w");          /* ouvrir en écriture */
[14]   strcpy(string,"Ceci est une ligne.");
[15]
[16]  for (index = 1;index <= 10;index++)
[17]       fprintf(fp,"Numéro de ligne  %d \t%s \n",index,string);
[18]
[19]   fclose(fp);                          /* Fin - Fermer le fichier */
[20] }
```

Ce programme commence, dans sa ligne 5, par l'intégration du fichier Include STDIO.H. Ensuite, sont déclarées quelques variables dont une variable de pointeur dirigée sur le nouveau type de données *FILE*. Le type de données *FILE* est défini dans STDIO.H, et sert à décrire la structure des fichiers MS-DOS.

Pour gérer des fichiers, le langage C requiert un pointeur dirigé sur le fichier à traiter. Ce pointeur doit être du type *FILE*.

Suite à la déclaration d'une variable de type *entier* et d'une variable de type *chaîne*, nous ouvrons le fichier LIGNES.TXT qui se trouve dans le répertoire principal de notre disque dur, à l'aide de la fonction *fopen()*. En guise de résultat, cette fonction retourne un pointeur du type de données FILE. Dans la ligne 13, nous attribuons au pointeur de fichier *fp* la valeur retournée par *fopen()*, par :

```
fp = fopen("LIGNES.TXT","w");
```

Dans la suite du programme, vous pouvez dès lors vous adresser à ce fichier par l'intermédiaire de ce pointeur. La fonction *fopen()* s'attend à recevoir comme premier argument le nom du fichier, et comme second argument le type de fichier qui en définit l'accès. Notez bien que les deux arguments sont des constantes alphanumériques (de type chaîne), et qu'il faut donc les écrire entre guillemets. Le type de fichier *w* (= write) définit le type d'accès ("écrire"). N'oubliez pas que, par cette déclaration, le contenu d'un éventuel fichier existant qui porte le même nom sera détruit dès ouverture du fichier !

Le fichier est ouvert. Dans la ligne 14, une chaîne de caractères est copiée dans la variable alphanumérique *string*, et celle-ci est ensuite écrite dix fois de suite dans le fichier ouvert au moyen d'une boucle de type *for*. Par la même occasion, un numéro est attribué à chaque ligne. La fonction *fprintf()* utilisée en ligne 15 (fprintf = file print formatted) utilise comme premier argument le pointeur de fichier, mais à par cela, elle est identique à la fonction *printf()* que vous connaissez. Nous pouvons donc formater la sortie et placer une marque de tabulation à l'aide '*t*'. Dans une dernière phase, le fichier qui vient d'être créé (LIGNES.TXT) est refermé à l'aide de la fonction *fclose()*. Dans ce dernier cas, l'argument est également un pointeur de fichier.

Dans le petit exemple précédent, la dernière instruction n'est pas obligatoire. En effet, à la fin d'un programme, tous les fichiers sont automatiquement refermés avant de revenir dans le système d'exploitation. Il vaut toutefois mieux prendre tout de suite les bonnes habitudes et ne rien laisser au hasard : les fichiers ouverts doivent être fermés avant de quitter un programme.

Lancez le programme, et chargez ensuite le fichier LIGNES.TXT dans l'éditeur de votre compilateur. Voyez qu'il contient effectivement dix lignes munies d'un numéro d'ordre, et qu'il n'y a aucune difficulté à charger ce fichier. La fonction *fprintf()* a donc généré un pur fichier ASCII, c'est-à-dire un fichier qui ne contient que des caractères du jeu ASCII qui peuvent être visualisés.

12.1.2. Ajouter des caractères

Dans l'exemple précédent, vous avez vu comment il est possible d'écrire des chaînes dans un fichier. Le programme CHAROUT.C montre la sortie orientée caractères dans un fichier.

Saisissez ce programme, et lancez-le. Observez ensuite le contenu du fichier LIGNES.TXT. Vous constaterez que dix lignes supplémentaires ont été ajoutées au fichier. Rechargez ensuite le programme.

Listing 49 : Le programme CHAROUT.C

```
[1 ] /* Chapitre 12.1.2 */
[2 ] /* Nom du programme: CHAROUT.C  */
[3 ] /* Exemple d'ajout de données à la fin d'un fichier */
[4 ] /* et d'écriture de caractères dans un fichier */
[5 ] #include <stdio.h>
[6 ] #include <string.h>
[7 ] #define FICHIER "C:\TC\LIGNES.TXT"
[8 ]
[9 ] void main()
[10] {
[11]   FILE *pointer;
[12]   char nouveau[35];
[13]   int index,compteur;
[14]
[15]   strcpy(nouveau,"Nouvelle ligne.");
[16]   pointer = fopen(FICHIER,"a");        /* ouvrir pour ajouter */
[17]
[18]   for (compteur = 1;compteur <= 10;compteur++)
[19]   {
```

```
[20]        for (index = 0;nouveau[index];index++)
[21]            putc(nouveau[index],pointer);      /* renvoi de caractères */
[22]        putc('\n',pointer);                    /* Saut de ligne */
[23]    }
[24]    fclose(pointer);                           /* Fermer le fichier */
[25] }
```

A la différence de ECRIS.C, le type d'accès au fichier LIGNES.TXT spécifié dans la ligne 16 est *a* (ajouter =, en anglais append). De plus, la constante symbolique FICHIER est utilisée à la place du nom du fichier.

La chaîne de caractères qui compose la variable *nouveau* de type chaîne est ensuite renvoyée par une boucle *for* imbriquée, écrite dans les lignes 18 à 23. La boucle externe est parcourue dix fois, et la boucle interne, répartie sur les lignes 20 et 21, est parcourue jusqu'à ce que soit atteinte la fin de la chaîne de caractères "*nouveau*". Comme vous le savez depuis le chapitre 8, l'expression *nouveau[index]* est vraie aussi longtemps que le symbole de fin de chaîne (*NULL*) n'est pas atteint.

Dans la boucle interne les caractères de "*nouveau*" sont ajoutés à l'extrémité du fichier existant. Ensuite, c'est le symbole de saut de ligne qui est écrit dans le fichier, et la boucle externe effectue un nouveau cycle. Après dix cycles, la boucle externe s'achève.

Nous utilisons ici, dans les lignes 21 et 22, la fonction *putc()*, qui écrit des caractères dans un fichier, en utilisant comme premier argument le caractère à écrire, et comme deuxième argument le pointeur dirigé sur le fichier.

12.2. Lecture d'un fichier

Maintenant que vous connaissez la technique d'écriture de caractères et de chaînes dans un fichier, nous allons voir, comment en C, l'on programme la lecture de données dans un fichier. Le programme qui suit renvoie à l'écran le contenu de notre fichier texte LIGNES.TXT. Il lit le fichier !

12.2.1. Lecture de caractères

Listing 50 : Le programme LECTURE.C

```
[1 ] /* Chapitre 12.2.1 */
[2 ] /* Nom du programme : LECTURE.C */
[3 ] /* Exemple de lecture de caractères dans un fichier */
[4 ]
[5 ] #include <stdio.h>
[6 ]
[7 ] main()
[8 ] {
[9 ]   FILE *lignes;
[10]   char c;
[11]
[12]   lignes = fopen("lignes.txt","r+b");    /* ouvrir en lecture */
[13]
[14]   if (lignes == NULL) printf("Fichier inexistant\n");
[15]                                  /* Fichier inexistant      */
[16]   else
[17]     {
[18]     do
[19]       {                          /* Exécute                 */
[20]         c = getc(lignes);        /* Extrait des caractères du
                                          fichier (par lecture)  */
[21]           putchar(c);            /* Affichage sur l'écran   */
[22]       } while (c != EOF);        /* jusqu'à l'extrémité du fichier (= EOF) */
[23]     }
[24] fclose(lignes);
[25] }
```

Le programme proprement dit commence à nouveau par l'intégration de *STDIO.H*, à l'aide de l'instruction *#include*. Après quelques déclarations de types, le fichier LIGNES.TXT est ouvert en mode *r* (= read, en anglais). La ligne 14 contient une nouveauté : nous y vérifions si le fichier existe. Si le fichier n'existe pas, le pointeur de fichier se voit attribuer le symbole de fin de chaîne, défini dans le fichier d'en-tête *STDIO.H* comme étant la constante symbolique *NULL*.

Lorsque le pointeur de fichier est égal à NULL, cela signifie donc que le fichier spécifié n'a pas pu être ouvert parce qu'il n'a pas été trouvé, ou parce qu'il n'existe pas.

Si le fichier LIGNES.TXT existe, nous voyons que sera parcourue la boucle *do..while* de la ramification *else* de la structure *if* (lignes 18 à 22). Cette boucle s'arrête dès qu'elle atteint le symbole de fin de fichier défini sous le nom *EOF* (End Of File = fin de fichier) dans le fichier d'en-tête. Les caractères sont d'abord

lus dans le fichier par l'intermédiaire de la fonction *getc()* de la ligne 20, puis, en ligne 21, ils sont renvoyés à l'écran au moyen de *putchar()*.

Avant de poursuivre, nous nous permettons d'insister une fois encore sur l'importance de la déclaration correcte des types de données. La fonction *getc()*, qui retourne à la partie appelante du programme une valeur de type *char*, en est un exemple supplémentaire. Dans l'exemple ci-dessus, cette valeur est attribuée à la variable *c*, qui a été déclarée en ligne 10 comme variable de type *char*. Mais la variable *c* est aussi utilisée en ligne 22 pour détecter le symbole de fin de fichier : la constante symbolique *EOF*. Mais la valeur de cette constante est *-1*. C'est en cela que réside le danger. Si *c* avait été déclarée comme *unsigned char* (type de données sans signe), le programme ne serait pas en mesure de reconnaître la fin d'un fichier, et son fonctionnement serait infini. En effet, si la variable *c* était un type de données sans signe, elle ne pourrait jamais prendre la valeur *-1* !

Modifiez maintenant le nom du fichier dans l'instruction *fopen*. Utilisez le nom d'un fichier qui n'existe pas dans le répertoire courant. Lancez le programme. Vous verrez que le test "*NULL*" fonctionne.

12.2.2. Lecture de mots

Dans le précédent programme de démonstration, vous avez vu comment, en C, des données peuvent être lues caractère par caractère dans un fichier. Le programme LISTEXTE.C montre comment extraire des chaînes de caractères ou mots d'un fichier.

Listing 51 : Le programme LISTEXTE.C

```
[1 ] /* Chapitre 12.2.2 */
[2 ] /* Nom du programme : LISTEXTE.C */
[3 ] /* Exemple de lecture de mots dans un fichier */
[4 ]
[5 ] #include <stdio.h>
[6 ]
[7 ] main()
[8 ] {
[9 ]   FILE *fp1;
[10]   char unmot[100];
[11]   char c;
[12]
[13]   fp1 = fopen("lignes.txt","r");
[14]   do
```

```
[15]   {                                  /* exécute */
[16]   c = fscanf(fp1,"%s",unmot);/* Extrait une chaîne du fichier */
[17]   printf("%s\n",unmot);              /* Affiche à l'écran */
[18]   } while (c != EOF);                /* tant que fin de fichier non atteinte */
[19]
[20]   fclose(fp1);
[21] }
```

Lorsque vous aurez chargé LISTEXTE.C, vous constaterez que le programme est très semblable à celui de l'exercice précédent. La différence réside dans le fait qu'ici, en ligne 16, c'est la fonction *fscanf()* qui est utilisée. Cette fonction se distingue de *scanf()* par le fait qu'elle assure une lecture dans un fichier, et non de données entrées au clavier. C'est pourquoi il faut ici spécifier comme premier argument le pointeur dirigé sur le fichier. Suivent ensuite, de même que dans *scanf()*, les descripteurs de champs et les variables dans lesquels les données extraites seront transmises. Dès que la fin de fichier est atteinte, *fscanf()* renvoie la valeur *EOF*. Ainsi, l'expression

```
c = fscanf(fp1, "%s",unmot);
```

qui se trouve dans la ligne 16 permet d'extraire de *fp1* une chaîne qui sera transcrite dans la chaîne *unmot*, puis de mémoriser le résultat de cette opération dans la variable *c*.

Comme vous pouvez le constater dans l'instruction de boucles de la ligne 17, la chaîne lue est renvoyée avant que l'instruction suivante ne localise la fin du fichier. Il en résulte que le dernier mot lu sera toujours affiché deux fois.

Comment pouvons-nous résoudre ce problème. Modifiez le programme LISTEXTE.C à l'aide des lignes que nous vous proposons ci-dessous, puis sauvegardez-le sous le nom LISBIEN.C.

Extrait de programme - Listing 52 : Le programme LISBIEN.C

```
[17] do
[18] {
[19] c = fscanf(fp1,"%s",unmot);
[20] if (c != EOF)              /* Vérifier si fin de fichier atteinte,
                                   avant!! affichage */
[21]   printf("%s\n",unmot);
[22] } while (c != EOF);
```

Vous constatez que, cette fois, dans la ligne 20, le symbole de fin de fichier sera détecté avant l'affichage, et ce renvoi n'aura lieu que si la fin du fichier n'est pas encore atteinte.

Lancez le programme. Vous constatez cette fois que la visualisation du contenu du fichier fonctionne correctement.

12.2.3. Lecture de lignes

L'affichage produit par les programmes que nous avons présentés dans les sections suivantes n'est pas très pratique. Il vaudrait mieux que le fichier soit renvoyé ligne par ligne. Le programme LISLIGN.C montre comment l'on peut atteindre cet objectif.

Listing 53 : Le programme LISLIGN.C

```
[1 ] /* Chapitre 12.2.3 */
[2 ] /* Nom du programme :LISLIGN.C */
[3 ] /* Lit des lignes dans un fichier */
[4 ]
[5 ]
[6 ] #include <stdio.h>
[7 ]
[8 ] void main()
[9 ] {
[10]   FILE *fp1;
[11]   char ligne[80],fichier[40];
[12]   char *c;
[13]
[14]   printf("Entrez le nom du fichier -> ");
[15]   c = gets(fichier);
[16]   fp1 = fopen(fichier,"r");
[17]
[18]   do
[19]   {
[20]     c = fgets(ligne,80,fp1);   /* Extrais une ligne du fichier */
[21]     if (c != NULL)             /* si pas fin de fichier, */
[22]       printf("%s",ligne);      /* renvoie la ligne à l'écran*/
[23]   } while (c != NULL);         /* jusqu'à ce que le pointeur *c soit
                                       dirigé sur NULL  */
[24]                                /* c'est-à-dire que la fin du fichier
                                       est atteinte    */
[25]   fclose(fp1);
[26] }
```

Dans ce programme, c'est une nouvelle fonction qui est utilisée (en ligne 20) pour lire une ligne : il s'agit de *fgets()*. Cette fonction lit une ligne, y compris le caractère de saut de ligne, et la transcrit dans une chaîne. Cette chaîne constitue le premier argument de la fonction. Comme deuxième argument suit le nombre maximal de caractères que cette chaîne peut comporter, et le troisième argument est le pointeur dirigé sur le fichier.

L'instruction

```
c = fgets(ligne,80,fp1);
```

extrait par conséquent des caractères d'un fichier jusqu'à ce que soit détecté un caractère de saut de ligne, ou que soit atteint le nombre maximal de caractères pouvant être transcrits, diminué de 1.

La fonction *gets()* qui, en ligne 15, est utilisée pour charger le nom d'un fichier est "de la même famille" que fgets(). La différence réside dans le fait que l'entrée provient ici du périphérique d'entrée standard : le clavier. C'est pourquoi le troisième argument de *fgets()*, le pointeur dirigé sur un fichier, est ici sans objet. L'instruction

C = gets(fichier,40); (Quick C)

C = gets(fichier); (Turbo C)

enregistre par conséquent des caractères saisis au clavier jusqu'à ce que 39 caractères aient été entrés, ou que l'utilisateur ait appuyé sur <Return>. Ces caractères seront ensuite mémorisés dans la variable de chaîne "*fichier*". La syntaxe requise par les compilateurs Quick C et Turbo C est ici différente. Alors que le compilateur de chez Microsoft demande la longueur de la chaîne en guise de second argument, ceux de chez Borland n'ont pas besoin de cette information. Ils la déterminent à partir de l'unique argument qu'ils exigent : la variable de type chaîne.

Lancez le programme et analysez l'affichage obtenu sur votre écran.

12.3. Ouverture et lecture de fichiers quelconques

A l'exception de notre dernier exemple, le nom du fichier à ouvrir était toujours intégré dans les programmes. Le programme FICHIER.C qui suit vous montre que le nom du fichier à ouvrir peut être saisi par l'utilisateur pendant l'exécution du programme. Le fichier ainsi ouvert sera ensuite renvoyé à l'écran.

Listing 54 : Le programme FICHIER.C

```
[1 ] /* Chapitre 12.3  */
[2 ]
[3 ] /* Nom du programme : FICHIER.C  */
[4 ] /* Exemple d'interrogation, d'ouverture et de lecture d'un */
[5 ] /* fichier quelconque */
[6 ]
[7 ] #include <stdio.h>
[8 ]
[9 ] main()
[10]   {
[11]   FILE *fp1;
[10]   char ligne[80],nomfichier[64];
[11]   char *c;
[12]
[13]   printf("Entrez le nom du fichier -> ");
[14]   scanf("%s",nomfichier);          /* Lis le nom de fichier choisi */
[15]   fp1 = fopen(nomfichier,"r");
[16]   if (fp1 == NULL)
[17]       printf("\n\n\nFichier non trouvé -  Fin du programme");
[18]   else
[19]   {                                /* Début de ELSE */
[20]       do
[21]       {
[22]           c = fgets(ligne,80,fp1);     /* lecture et */
[23]           if (c != NULL)               /* pas fin de fichier */
[24]               printf("%s",ligne);      /* Renvoi de lignes */
[25]       } while (c != NULL);             /* jusqu'à la fin du fichier */
[26]   }                                    /* Fin de ELSE */
[27]   fclose(fp1);
[28] } /* Fin de main */
```

Dans ce listing, rien ne vous est inconnu. Nous nous passerons donc de commentaires, et vous laissons tout le temps pour analyser vous même la logique du programme.

12.4. Transmission de noms de fichiers à main()

Il est évidemment possible de rédiger les programmes C de manière à ce que des arguments puissent leur être communiqués à l'occasion de leur lancement. Comme vous le savez, la fonction *main()* est activée immédiatement après le lancement du programme, en guise de programme principal, et gère le déroulement du programme par appel à d'autres fonctions, jusqu'à la fin du programme. Par conséquent, les paramètres spécifiés derrière le nom du programme dans l'environnement du système d'exploitation font office d'arguments transmis à *main()*.

Le programme MAINC.C nous montre la transmission d'arguments à un programme lors de son lancement, en prenant comme exemple un traitement de fichiers.

Listing 55 : Le programme MAINC.C

```
[1 ] /* Chapitre 12.4*/
[2 ] /* Nom du programme: MAINC.C*/
[3 ] /* Exemple de transmission d'arguments à main()  */
[4 ] /* dans l'environnement DOS */
[5 ] #include "stdio.h"
[6 ]
[7 ]
[8 ] main(int argc, char *argv[])
[9 ] {
[10]
[11]   char ligne[80];
[12]   char *c;
[13]   FILE *fpt;
[14]
[15]   if (argc == 1)     /* vrai, si aucun argument n'est transmis */
[16]   {
[17]     printf("Erreur - N'est pas un nom de fichier -
               Fin du programme....");
[18]     exit(0);          /* Quitter le programme */
[19]   }
[20]   if ((fpt=fopen(argv[1],"r"))== NULL)
[21]       printf ("Erreur lors de l'ouverture du fichier");
[22]
[23]
[24]   do
[25]   {
[26]       c = fgets(ligne,80,fpt);
[27]       if ( c != NULL)
[28]       printf("%s",c);
[29]    } while (c != NULL);
[30]  }
```

Dans la ligne 8, vous voyez que les parenthèses de la fonction *main()* renferment deux arguments. Cela signifie que *main()* s'attend à recevoir des arguments lors de son activation (qui correspond toujours au lancement du programme). La variable entière *argc* contient, dès le lancement du programme le nombre des paramètres transmis, et le pointeur de chaîne **argv[]* contient les paramètres de type chaîne qui ont été associés au programme, *argv[0]* contenant toujours le nom du programme. Dans *argv[1]* est mémorisé le premier paramètre, dans *argv[2]* est mémorisé le second, etc... La valeur de *argc* est *1* lorsqu'aucun paramètre n'est associé au lancement du programme, et pour chaque paramètre complémentaire cette valeur est incrémentée de 1.

Avant que ce programme ne tente d'ouvrir un fichier, il vérifie, en ligne 15, si un paramètre lui a été associé. C'est le cas si la valeur de argc est supérieure à 1. Dans le cas contraire, un message d'erreur est renvoyé (ligne 17), et le programme s'achève en ligne 18 par l'instruction

```
exit(0);
```

Exit() achève toujours la fonction appelante. Etant donné que, dans notre cas, il s'agit de la fonction *main()*, c'est tout le programme qui sera achevé. Il est possible de communiquer à la fonction de niveau supérieur une valeur conditionnant l'achèvement. Le rôle de fonction de niveau supérieur peut évidemment être tenu par MS-DOS.

L'usage s'est répandu de transmettre à la fonction *exit()* la valeur 0 lorsque la fonction est quittée normalement, et des valeurs différentes de 0 en cas d'erreur. Les valeurs retournées par *exit()* au système d'exploitation peuvent être interrogées dans des fichiers batch sous forme d'Errorlevel.

Dans la ligne 20, dans laquelle nous n'arrivons que si un paramètre a effectivement été transmis au programme, le programme interprète le premier argument spécifié comme s'il s'agissait d'un nom de fichier, et tente d'ouvrir ce fichier en lecture. L'instruction

```
fpt=fopen(argv[1], "r");
```

signifie par conséquent : "interprète le premier paramètre spécifié pour le lancement du programme comme nom de fichier, puis tente d'ouvrir ce fichier en lecture."

La boucle de renvoi qui suit est identique à celle que contenaient les programmes précédents, et ne nécessite donc aucune explication supplémentaire.

Compilez ce programme, puis quittez le compilateur. Entrez maintenant la suite de commandes :

```
mainc lignes.txt
```

Vous voyez que le contenu du fichier LIGNES.TXT est renvoyé à l'écran. Entrez la même instruction, avec le nom d'un fichier qui n'existe pas. Vous constatez que le programme fonctionne parfaitement, compte-tenu des paramètres qui lui ont été transmis. Entrez maintenant le nom du programme sans spécifier de paramètres. Le

fonctionnement du programme correspond, ici aussi, à l'idée que nous nous en étions faite.

12.5. Impression du contenu d'un fichier

Jusque-là, vous ne savez pas encore comment transférer le contenu d'un fichier sur papier. Le programme qui suit vous montre la technique à utiliser.

Listing 56 : Le programme IMPRFICH.C

```
[1 ] /* Chapitre 12.5 */
[2 ] /* Nom du programme : IMPRFICH.C */
[3 ] /* Exemple d'affichage et impression d'un fichier */
[4 ]
[5 ] #include <stdio.h>
[6 ]
[7 ] main()
[8 ] {
[9 ]    FILE *fichier,*imprimante;
[10]   char c;
[11]
[12]   fichier=fopen("lignes.txt","r");    /* Lecture du fichier d'entrée */
[13]   imprimante = fopen("PRN","w");      /* Ecriture sur imprimante */
[14]
[15]   do
[16]   {
[17]       c = getc(fichier);          /* Lire caractère dans le fichier */
[18]       if (c != EOF)
[19]       {
[20]           putchar(c);             /* sortie caractères à l'écran */
[21]           putc(c,imprimante);     /* sortie caractères sur imprimante */
[22]       }
[23]   } while (c != EOF);
[24]   fclose(fichier);
[25]   fclose(imprimante);
[26] }
```

Vous serez surpris de la simplicité de la technique d'impression d'un fichier. Le fichier LIGNES.TXT est ouvert en lecture, dans la ligne 12, suivant la méthode habituelle. Nous ouvrons simultanément un fichier dont le nom est *PRN*, en mode "w" (ouverture en écriture). Etant donné que *PRN* est le nom attribué par le système d'exploitation à l'imprimante connectée sur le premier port parallèle, et que MS-DOS gère les périphériques de la même façon que les fichiers, l'écriture dans le fichier *PRN* se solde par l'impression des données sur papier.

La partie "sortie" du programme est donc composée d'une simple boucle (lignes 15 à 23), dans laquelle le contenu du fichier-source est extrait caractère par caractère, puis transmis dans le "fichier de sortie", c'est-à-dire l'imprimante. Dès que la fin du fichier-source est atteinte, le programme referme les deux fichiers ouverts.

Mettez votre imprimante en marche, alimentez-la en papier, puis lancez notre programme. Après quelques instants, le contenu du fichier LIGNES.TXT sera imprimé.

12.6. Récapitulation

En C, chaque traitement de fichier commence par l'ouverture du fichier concerné au moyen de la fonction *fopen()*, qui demande deux arguments : le nom du fichier et le mode d'accès. Les modes d'accès possibles sont "*w*" pour l'écriture, "*r*" pour la lecture, et "*a*" pour ajouter des données complémentaires. La valeur retournée par la fonction est attribuée à un pointeur du type *FILE*. Chaque accès au fichier s'effectue alors par l'intermédiaire du pointeur associé au fichier. La valeur du pointeur associée au fichier est *NULL* lorsque ce fichier ne peut être ouvert. La constante symbolique *EOF* sert à détecter la fin d'un fichier. Pour fermer un fichier l'on utilise la fonction *fclose()*.

La fonction *fprintf()* permet de programmer des sorties structurées dans un fichier. L'écriture dans un fichier de caractères individuels est assurée par *putc()*. La fonction *puts()* écrit des chaînes non structurées sur l'écran. Il est possible d'extraire d'un fichier des caractères individuels, des chaînes ou des lignes entières. L'extraction de caractères est assurée par *getc()*, celle de chaînes ou de lignes est obtenue au moyen de *fgets()*.

La fonction *exit()* achève la fonction appelante et peut transmettre à la fonction de niveau directement supérieur une valeur. La valeur 0 signale une fin normale de la fonction. D'autres valeurs signifient qu'une erreur s'est produite avant l'achèvement de la fonction.

Pour imprimer un fichier, le contenu de celui-ci doit d'abord être lu, puis écrit dans le fichier PRN. PRN est le nom que le DOS associe à l'imprimante.

Exercices se rapportant au chapitre 12

Exercice 1 :
Ecrivez un programme qui invite son utilisateur à entrer le nom d'un fichier texte, puis qui ouvre ce fichier en lecture et renvoie son contenu à l'écran. Pour symboliser le nom du fichier et le pointeur qui lui est associé, vous utiliserez respectivement les variables globales *fichier* et *pointeurfichier*.

Programmez la fonction

```
get_nomfichier()
```

qui devra se charger de lire le nom du fichier.

Programmez une fonction

```
int open_fichier(char *nomfichier, char statut)
```

qui assume les tâches suivantes :

- Ouvrir le fichier transmis par l'intermédiaire de *nomfichier* dans le mode indiqué par la variable statut.
- Renvoyer un message d'erreur dès qu'un défaut se présente.
- Renvoyer la valeur 0 dès qu'une erreur surgit, et la valeur 1 dans le cas contraire.

Programmez pour la sortie du fichier, la fonction

```
int output(FILE *pointeurfichier)
```

qui devra assumer les tâches suivantes :

- Visualiser à l'écran le contenu du fichier indiqué par le pointeur.
- Renvoyer la valeur 1 à la fonction appelante dès lors que le fichier a pu être lu jusqu'à son extrémité.

La structure du programme principal devrait présenter l'allure suivante :

```
main()
{
  get_nomfichier();
  if (open_fichier(fichier,"r"))
    output(pointeurfichier);
}
```

Exercice 2 :
Modifiez le programme précédent de manière à ce qu'un nom de fichier ne soit demandé à l'utilisateur que si celui-ci n'a pas déjà été spécifié lors de l'activation du programme pour être transmis immédiatement à *main()*.

13 Types de données complexes

Dans ce chapitre vous apprendrez comment créer des structures de données complexes à l'aide des mots clé *struct*, *typedef* et *union*, ou comment définir de nouveaux types de données à partir des types de données existants. Vous écrirez, rechercherez, lirez et modifierez des enregistrements structurés, et découvrirez par la même occasion les notions fondamentales qui régissent la programmation des bases de données. Enfin, vous apprendrez à déplacer le pointeur associé à un fichier à l'intérieur de ce fichier.

13.1. Structures

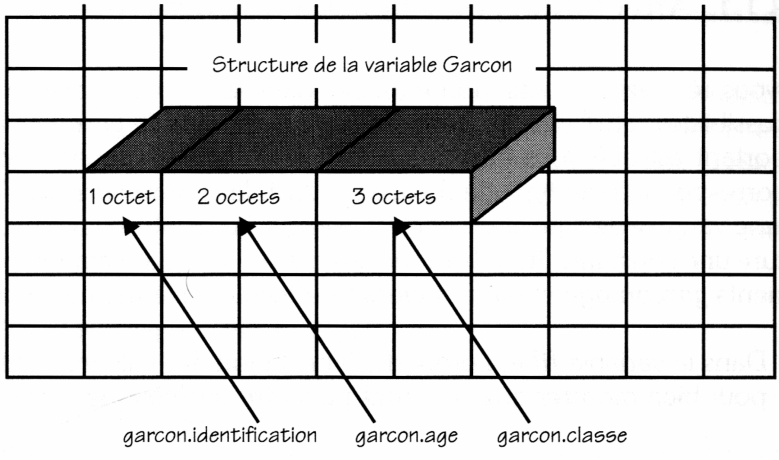
On appelle structures les types de données définis par le programmeur à partir de types de données simples.

Il est donc possible de définir des types de données plus complexes que ceux que nous avons connus jusque-là. Une structure est toujours une combinaison de types de données et de structures définis précédemment. Le meilleur moyen pour comprendre ce qu'est une structure nous semble une fois de plus être l'étude sur l'exemple. Le programme STRUCT1.C qui suit est volontairement simplifié au maximum, et nous permettra d'aborder les nouveaux types de données. Nous déclarons ici une structure décrivant les caractéristiques des élèves d'un établissement scolaire. Des valeurs sont ensuite attribuées à cette structure qui les renverra à l'écran.

Listing 57 : Le programme STRUCT1.C

```
[1 ] /* Chapitre 13.1 */
[2 ] /* Nom du programme : STRUCT1.C */
[3 ] /* 1. Exemple de définition d'une structure */
[4 ]
[5 ] main()
[6 ] {
[7 ]
[8 ]   struct
[9 ]       {
[10]       char identification;          /* Initialise */
[11]       int age;                      /* Age de l'enfant */
[12]       int classe;                   /* Classe fréquentée */
[13]       } garcon, fille;
[14]
[15]      garcon.identification = 'R';   /* Le garçon d'identification R */
[16]       garcon.age = 15;              /* a 15 ans et */
[17]       garcon.classe = 3;            /* est en classe de 3ème */
[18]
[19]       fille.age=garcon.age-1;       /* 1 an de moins  */
[20]       fille.classe=4*;              /* est en classe de 4ème */
[21]       fille.identification = 'H';   /* et identification H */
[22]
[23]       printf("%c a %d ans, est en %dème\n",
[24]          fille.identification, fille.age, fille.classe);
[25]
[26]       printf("%c a %d ans, est en %dème\n",
[27]          garcon.identification, garcon.age, garcon.classe);
[28] }
```

Ce programme de démonstration commence réellement en ligne 8, par la définition d'une structure. Le mot clé *struct* est suivi de trois variables de type simple, réunies en un bloc au moyen d'accolades. L'accolade fermante est suivie de deux noms de variables : *garcon* et *fille*. Conformément à la définition de la structure, la variable *garcon* est composée de trois éléments : les variables *identification*, *age*, et *classe*, dont l'ensemble constitue une structure, et par voie de conséquence un nouveau type de données. Chacune de ces variables est liée à la variable garcon, et chacune peut prendre des valeurs du type approprié.



La structure de la variable garcon

Les mêmes remarques s'appliquent à la variable *fille*. En fait, nous venons de définir six variables.

Voyons d'un peu plus près la variable *garcon*. Elle regroupe trois variables simples qui peuvent être utilisées à tout endroit d'un programme C si le type de données le permet. Considérons par exemple la variable entière *age*. Vous pouvez l'utiliser pour faire des calculs, comme variable d'incrémentation, ou encore dans le cadre d'opérations entrée/sortie.

Mais comment pouvez-vous vous adresser à la variable simple *age* qui, pour l'instant, est indissociable de la variable *garcon* ? Et bien il suffit d'utiliser les deux noms, en les séparant par un point. Le nom de niveau supérieur (c'est-à-dire *garcon*) doit figurer en début de notation. Par conséquent, *garcon.age* est le nom de variable complet du champ *age* de la variable de structure *garcon*. Le nom *age* ne peut en aucun cas être utilisé comme désignation de variable autonome, puisqu'il ne constitue qu'une définition partielle de la structure.

13.1.1. Affectations de valeurs à des structures

Si vous vous référez à la définition énoncée dans la section précédente, vous avez la possibilité d'attribuer des valeurs aux trois champs de *garcon* et *fille*. Le plus important est que vous affectiez aux divers éléments de la structure des valeurs correspondant au type de données. Ainsi, le champ *garcon.identification* est une variable de type *char*, et c'est pourquoi vous attribuez à cet élément de structure une constante littérale monocaractère : "R". L'affectation de valeurs aux éléments *garcon.age* et *garcon.classe* s'effectue de la même façon.

Dans la variable *fille*, l'ordre de l'affectation des valeurs a été modifié pour bien montrer que cet ordre peut être quelconque.

13.1.2. Affichage de champs de structures

Dès que des valeurs ont été attribuées aux six variables, celles-ci peuvent être utilisées à volonté. Par souci de simplicité, dans ce premier exemple de programme, le contenu des variables est renvoyé à l'écran à l'aide de l'instruction *printf*. Vous pouvez voir que le renvoi de champs de structures ne diffère en rien du renvoi des variables courantes.

Lorsque vous lancez le programme de démonstration ci-dessus, vous voyez le contenu des variables de structure *garcon* et *fille* sur votre écran.

Voici l'affichage retourné par le programme STRUCT1.C

```
H a 14 ans, est en 4ème
R a 15 ans, est en 3ème
```

L'utilisation de types de données définis par le programmeur est une excellente technique qui permet de regrouper des données, et par conséquent de rendre les codes des programmes plus clairs. Dans la suite de ce chapitre, vous verrez que les structures permettent de résoudre des problèmes très complexes.

13.2. Un tableau de structures

Dans le programme qui suit, nous utilisons une structure très semblable à celle employée dans STRUCT1.C. La différence réside dans le fait que cette structure est attribuée à un tableau de douze éléments (dans la ligne 12). C'est pourquoi le programme renferme 12 * 3 = 36 variables. Grâce à cet exemple, vous apprendrez qu'il suffit d'une simple affectation pour copier le contenu de structures.

Listing 58 : Le programme STRUCT2.C

```
[1 ] /* Chapitre 13.2 */
[2 ] /* Nom du programme : STRUCT2.C */
[3 ] /* Déclaration d'un tableau de structures - Attribution de structure */
[4 ]
[5 ] main()
[6 ] {
[7 ]   struct
[8 ]       {
[9 ]       char identification;
[10]       int age;
[11]       int classe;
[12]       } enfants[12];
[13]
[14]   int index;
[15]
[16]   for (index = 0;index < 12;index++)
[17]   {
[18]       enfants[index].identification = 'A' + index;
[19]       enfants[index].age = 16;
[20]       enfants[index].classe = 1;
[21]   }
[22]
[23]   enfants[3].age = enfants[5].age = 17;
[24]   enfants[2].classe = enfants[6].classe = 3;
[25]   enfants[4].classe = 2;
[26]
[27]   enfants[10] = enfants[4];    /* attribution d'une structure  */
[28]
[29]   for (index = 0;index < 12;index++)
[30]       printf("%c a %d ans, est en % de\n",
[31]               enfants[index].identification, enfants[index].age,
[32]               enfants[index].classe);
[33] }
```

Observez dans un premier temps la boucle *for* de la ligne 16. Vous voyez qu'en un cycle de la boucle, toutes les valeurs nécessaires sont transmises à chaque enfant. Les lignes 23 à 27 qui suivent montrent différentes possibilités d'affectation de valeurs. Vous constatez que l'attribution multiple est tout aussi possible

que la copie d'une structure complète dans une autre. A la fin de ce programme de démonstration, le contenu du tableau est renvoyé à l'écran.

L'apprentissage de la technique étant notre souci primordial, les valeurs que nous attribuons aux variables ne présentent aucun intérêt : elles ne sont utilisées qu'aux fins de démonstration. Dans un programme appliqué à la pratique, de telles valeurs pourraient par exemple être extraites d'un fichier et attribuées à des variables.

13.3. Pointeurs et structures

Le programme STRUCT3.C montre le travail avec des pointeurs dirigés sur des structures. Vous verrez comment on peut programmer l'accès à une structure à l'aide de pointeurs dirigés sur cette structure. Par la même occasion, vous aborderez les premières notions de programmation des systèmes de gestion de bases de données.

Dans notre exemple, nous définissons une structure décrivant les caractéristiques d'un employé. Le programme prévoit diverses possibilités d'accès aux divers éléments de la structure.

Listing 59 : Le programme STRUCT3.C

```
[1 ] /* Chapitre 13.3 */
[2 ] /* Nom du programme : STRUCT3.C */
[3 ] /* Structures, pointeurs et arithmétique de pointeurs  */
[4 ]
[5 ] main()
[6 ] {
[7 ]   struct
[8 ]   {
[9 ]   char identification;        /* Lettre d'identification */
[10]   int pnr;                    /* Numéro personnel */
[11]   long chiffre_affaire;       /* Chiffre d'affaire, entier, exprimé en
                                      centaines */
[12]   } employe[12],*pointeur,extra;
[13]
[14]   int index;
[15]
[16]   for (index = 0;index < 12;index++)    /* Affectation de valeurs */
[17]   {
[18]       pointeur = employe + index;
[19]       pointeur->identification = 'A' + index;
[20]       pointeur->pnr = 0 + index;
[21]       pointeur->chiffre_affaire = 100;
[22]   }
```

```
[23]
[24]    employe[3].pnr = employe[5].pnr = 34;
[25]    employe[2].pnr = employe[4].pnr + 10;
[26]    employe[0].chiffre_affaire = 2000;
[27]
[28]    for (index = 0;index < 12;index++)    /* Boucle de renvoi */
[29]    {
[30]    pointeur = employe + index;
[31]    printf( "Le numéro de %c est %d et son chiffre d'affaire est
                de %ld Francs\n",
[32]            (*pointeur).identification, employe[index].pnr,
[33]            ((pointeur->chiffre_affaire) * 100));
[34]    }
[35]    extra = employe[2];                    /* Affectations de structures */
[36]    extra = *pointeur;                     /* Affectations de structure */
[37] }
```

Vous voyez que, pour la structure définie dans les lignes 9 à 11, il est également déclaré un pointeur dans la ligne 12. "*pointeur*" est par conséquent un pointeur dirigé sur la structure. Si, pendant l'exécution du programme, *pointeur* était dirigé sur un autre type de données, une erreur en résulterait. La raison de ce comportement a été traitée en détail dans le chapitre 10. Toutefois, pour approfondir cette question, rappelons-en le principe.

En raison de la définition de la structure, le programme connaît le nombre d'octets dont elle a besoin en mémoire de travail. Comme vous le savez, un pointeur est toujours dirigé sur l'origine d'une variable. L'origine de la variable suivante dépend ensuite du nombre d'octets occupé par le type de données concerné. La variable *employe* occupe dans notre cas sept octets répartis en :

1 * 1 octet pour le type de données char, 1 * 2 octets pour le type de données int et 1 * 4 pour le type de données long.

Par conséquent, dès qu'un pointeur est dirigé sur *employe[0]*, la prochaine variable, *employe[1]* commencerait à 7 octets derrière le premier octet de *employe[0]*. La valeur du pointeur dirigé sur *employe[1]*, qui est en fait l'adresse de cette variable en mémoire de travail, serait donc supérieure de cette unité à l'adresse de *employe[0]*.

Dans ce programme, les premières affectations de valeurs sont effectuées par l'intermédiaire d'une boucle *for*, en ligne 16. Cette fois, c'est toutefois le pointeur "*pointeur*" qui est utilisé. L'instruction

```
pointeur = employe + index;
```

qui figure en ligne 18 fait en sorte que "*pointeur*" soit placé sur un autre champ du tableau *employe* après chaque cycle de la boucle. Vous avez à faire ici à un exemple typique de mathématique des pointeurs, sujet que nous avons déjà abordé dans le passé, et que nous allons approfondir dans la section suivante.

13.4. Arithmétique des pointeurs

Lorsque le pointeur "*pointeur*" est incrémenté de 1, cela ne signifie pas que "*pointeur*" soit maintenant dirigé sur le second octet de la variable *employe[0]*, mais que la valeur du pointeur est incrémentée de :

1 * longueur du type de données qui est celui de *employe*, exprimée en octets

Dans notre cas, le pas de l'incrémentation sera donc 7. Le pointeur sera ainsi effectivement dirigé sur la variable employe[1] qui suit. C'est là la raison pour laquelle un pointeur ne peut être dirigé que sur le type de données qui lui a été affecté au départ. Ce n'est qu'en se conformant à cette règle que la mathématique des pointeurs sera exempte d'erreurs.

Sachez également que le pointeur "*pointeur*" peut aussi être dirigé sur chaque élément qui entre dans la composition de la structure. L'endroit auquel commence un élément est à nouveau fonction du type de données, et plus particulièrement de l'allocation en mémoire d'une variable de ce type. Un opérateur est spécialement défini pour pouvoir pointer sur un élément d'une variable de structure à l'aide d'un pointeur : il s'agit de "->".

Lorsque nous écrivons

```
pointeur -> identification
```

cette syntaxe est synonyme de

```
(*pointeur).identification
```

qui est aussi équivalent à

```
employe[i].identification.
```

Si, à ce stade, vous vous heurtez à des difficultés de compréhension, nous vous conseillons vivement de réétudier le chapitre 10. Pour mémoire : n'oubliez pas que **pointeur* équivaut au contenu de la variable dont l'adresse est mémorisée dans "*pointeur*". Sachez également que dans l'expression

```
(*pointeur).identification
```

la priorité du point est supérieure à celle de l'astérisque qui symbolise le pointeur. C'est pourquoi les parenthèses sont indispensables. Une expression telle que **pointeur.identification* serait semblable à **(pointeur.identification)*, et ne produirait pas le résultat attendu.

Pour préciser cette notion, les trois méthodes d'accès au contenu d'une variable sont utilisées dans la boucle d'affichage finale, dans les lignes 28 à 34. Ce style de programmation n'est évidemment pas élégant, mais permet de revoir les diverses techniques d'adressage.

A la fin du programme, vous voyez deux exemples d'affectation de structures complètes, en ligne 35 par attribution de variables, et en ligne 36 par attribution au moyen d'un pointeur.

13.5. Structures nommées et imbriquées

Jusque ici, nous n'avons fait la connaissance que de structures simples. Dans cette section, vous verrez que des structures peuvent également être imbriquées. Cela signifie que, dans la définition d'une structure, vous pouvez utiliser des structures définies précédemment.

Il est possible de créer des structures contenant des dizaines d'éléments. C ne connaît aucune limite dans ce domaine. Suivant le problème posé, un grand nombre d'éléments peut être nécessaire. Il est toutefois préférable, dans ce cas, de ne pas définir tous les éléments en un seul niveau de structure. Il est davantage conseillé de construire une structure hiérarchique dans laquelle les divers éléments, qui pourront à nouveau être constitués de structures, sont organisés clairement. Vous verrez que, grâce à l'attribution de noms à des structures, le code du programme gagne sensiblement en lisibilité.

Le programme IMBRIQUE.C apporte la démonstration de cette technique. L'organisation des informations concernant le personnel y est affinée grâce à des structures définies à cet effet auxquelles est attribué un nom.

Listing 60 : Le programme IMBRIQUE.C

```
[1 ] /* Chapitre 13.5 */
[2 ] /* Nom du programme: IMBRIQUE.C */
[3 ] /* Exemple de structures imbriquées */
[4 ]
[5 ] main()
[6 ] {
[7 ]  struct personne
[8 ]  {
[9 ]    char nom[25];
[10]    int age;
[11]    char statut;                    /* m = marié, c = célibataire */
[12]  } ;
[13]
[14]  struct employe
[15]  {
[16]     int servicenr;                 /* numéro du service */
[17]     struct personne caracteristique;  /* caractéristiques
                                              personnelles */
[18]     char position[25];       /* Position occupée dans l'entreprise */
[19]  } personnel[53];            /* 53 employés au maximum */
[20]
[21]  struct employe directeur,autres;   /* autres variables */
[22]
[23]  directeur.servicenr = 1;
[24]  directeur.caracteristique.age = 54;
[25]  directeur.caracteristique.statut = 'm';
[26]  strcpy(directeur.caracteristique.nom,"Margot Monnet");
[27]  strcpy(directeur.position,"titulaire");
[28]
[29]  autres.caracteristique.age = 40;
[30]  autres.caracteristique.statut = 'm';
[31]  strcpy(autres.caracteristique.nom,"Martine Dalloi");
[32]  autres.servicenr = 0;
[33]  strcpy(autres.position,"néant");
[34]
[35]  personnel[1].caracteristique.age = 30;
[36]  personnel[1].caracteristique.statut = 'c';
[37]  strcpy(personnel[1].caracteristique.nom,"Michel Dupond");
[38]  strcpy(personnel[1].position,"Employé");
[39]  personnel[1].servicenr = 1;
[40]
[41]  personnel[7].caracteristique.age = 16;
[42]  personnel[12].servicenr = 2;
[43]
[44]  printf("Le collaborateur %s a %d ans, occupe la position %s",
```

```
[45]            personnel[1].caracteristique.nom,
                personnel[1].caracteristique.age,
      personnel[1].position);
[46]
[47]    printf("\net la position du directeur est %s.",directeur.position);
[48] }
```

La première structure contient trois éléments. Etant donné que sa définition n'est suivie d'aucun nom de variable, il n'existe dans l'immédiat pas de variable dont le type de données corresponde à cette structure. A son début, cette structure a toutefois reçu un nom : *personne*. Dans ce cas, l'on parle d'une structure nommée. Ainsi, le nom "*personne*" pourra être utilisé dans le programme à chaque fois que l'on souhaite se référer à cette structure. Il ne faut toutefois pas qu'il existe une autre variable portant ce même nom !

Un nouveau type de données vient d'être défini. Il s'appelle "*personne*", et peut être utilisé de la même façon que les types de données *int* ou *char*. La seule restriction provient du fait que, devant "*personne*", il vous faudra placer le mot clé *struct*. C'est à ce mot clé que le compilateur reconnaît que "*personne*" est une structure définie par le programmeur.

Dans la structure dont la définition est répartie sur les lignes 14 à 19, vous pouvez reconnaître un exemple d'organisation hiérarchique de structure. Son nom est *employe*, et les éléments qui le composent sont : *servicenr*, donnée de type *int*, *caracteristique*, donnée de type *struct personne*, et *position*, chaîne de 25 caractères. Etant donné que la structure "*personne*" est composée de trois éléments, la structure "*employe*" en renferme cinq.

Dans la ligne 19 est déclaré un tableau appelé *personnel*, contenant 53 champs dont le type de données est *struct employe*.

Une fois que le nouveau type de données *employe* est défini, l'on peut, dans la suite, lui attribuer d'autres variables. C'est ce qui se produit dans la ligne 21 avec l'instruction

```
struct employe directeur, autres;
```

pour les variables *directeur* et *autres*.

13.6. Affectations de valeurs et attribution de noms

Dans les lignes 23 à 27 du programme, une valeur est attribuée à chaque élément de la variable de structure "*directeur*". Pour le premier élément, *servicenr*, l'affectation est réalisée de la manière habituelle. Dans la suite, pour affecter un âge au directeur, il est fait appel à trois étapes :

① Il y a d'abord le nom de variable "*directeur*".

② Vient ensuite le nom de la structure "*personne*", c'est-à-dire "*caracteristique*".

③ Pour finir, l'élément désiré de "*caracteristique* ", soit dans notre cas "*age*", est ajouté à la suite des précédents.

Les divers éléments du nom doivent, dans cette notation, être séparés par un point. C'est de cette convention que résulte l'affectation

```
directeur.caracteristique.age = 54
```

La même méthode est utilisée dans la ligne 25 pour définir le statut du directeur. Les éléments *nom* et *position* sont des chaînes. C'est la raison pour laquelle, dans les lignes suivantes, les valeurs leur sont attribuées à l'aide de la fonction *strcpy()*.

Voici une convention qu'il convient de respecter lorsque l'on attribue un nom à des éléments d'une structure :

Les chaînes de noms que nous utilisons ici sont des noms de variables, et sont traités comme n'importe quel autre nom de variable.

Des valeurs sont également affectées à la variable *autres*. Nous avons une fois de plus modifié l'ordre d'affectation afin de prouver que l'attribution de valeurs aux divers éléments peut être réalisée dans un ordre quelconque.

En théorie, l'imbrication des structures n'est pas limitée en quantité. Ne perdez toutefois pas de vue qu'en multipliant les imbrications vous risquez de perdre la maîtrise du texte de votre programme. Vous pouvez également intégrer plusieurs fois une même structure dans une structure donnée. D'autre part, des structures peuvent renfermer des tableaux eux-mêmes composés de structures.

13.7. Le type de données union

Le programme UNION1.C va nous permettre de découvrir un type de données qui permet de mémoriser à une même adresse de la mémoire de travail des données de types différents. Le nom de ce nouveau type de données est *union*, et celui-ci permet de construire des structures de données complexes.

Dans le programme ci-dessous, le contenu d'une variable enregistrée en mémoire est renvoyé à l'écran sous deux formes distinctes : comme variable de type *entier*, d'une part, et sous forme de deux variables de type *char* en format hexadécimal d'autre part.

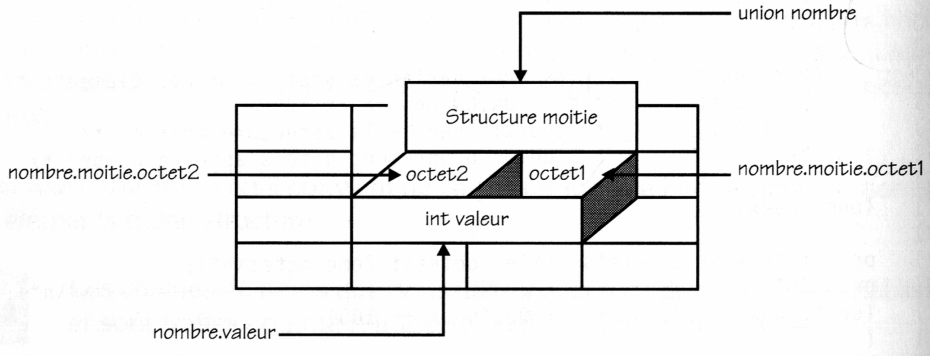
Listing 61 : Le programme UNION1.C

```
[1 ] /* Chapitre 13.7 */
[2 ] /* Nom du programme: UNION1.C */
[3 ] /* Exemple de déclaration et d'utilisation d'UNION en C */
[4 ]
[5 ] main()
[6 ] {
[7 ]   union
[8 ]   {                    /* premier et plus grand élément de l'UNION */
[9 ]   int valeur;          /* nombre */
[10]   struct
[11]       {
[12]       char octet1;     /* les deux variables sont aussi des éléments */
[13]       char octet2;     /* de l'UNION nombre et */
[14]       } moitie;        /* appartiennent à la structure moitie  */
[15]   } nombre;            /* L'union proprement dite s'appelle nombre */
[16]
[17]   long index;
[18]
[19]   printf("Valeur 2 octets \t 1er octet\t 2ème octet\n");
[20]   printf("-------------------------------------------------------\n");
[21]   for (index = 100;index < 1500;index += 100)
[22]   {
[23]      nombre.valeur = index;  /* Plage de mémoire de nombre -
[24]                                 2 octets occupés */
[25]
[26]      printf("%8x\t\t %6x\t\t %6x\t\t\n",nombre.valeur,
[27]             nombre.moitie.octet1,nombre.moitie.octet2);
[28]                              /* Affichage sous forme de valeurs
                                     hexadécimales */
[29]   }
[30]   printf("-------------------------------------------------------\n");
[31] }
```

Dans cet exemple, l'*union* définie dans les lignes 7 à 15 concerne deux éléments, et son nom est "*nombre*". Son premier élément s'appelle "*valeur*", et est une variable entière mémorisée à un certain endroit de la mémoire de travail où elle occupe deux octets. Sont ensuite définis deux autres éléments, de type *char*, qui sont de plus éléments de la structure *moitie*. Les deux éléments "*valeur*" et "*struct moitie*" nécessitent des espaces mémoire de même taille, et en raison de leur regroupement défini comme *union*, elles seront enregistrées dans la mémoire de travail à la même adresse. L'adresse de la structure *moitie* est donc la même que celle de la variable entière *valeur*. Il en résulte par conséquent que :

Une *union* permet de mémoriser des types de données différents au même emplacement physique de la mémoire.

Dans notre cas, vous pouvez donc mémoriser un nombre entier à l'emplacement de la mémoire où se trouve "*valeur*", et par la suite l'en extraire sous forme de deux valeurs d'un octet par l'intermédiaire des variables *octet1* et *octet2*.



Le modèle de mémorisation union

Dans les lignes 21 à 29 est programmée une boucle de renvoi qui, pour les valeurs comprises entre 100 et 1500, attribue d'abord une valeur entière à l'union *nombre.valeur*. L'instruction d'affichage

```
printf("%8x\t\t %6x\t\t
 %6x\t\t\n",nombre.valeur,nombre.moitie.octet1,nombre.moitie.octet2);
```

qui est programmée dans les lignes 26 et 27 renvoie d'abord la valeur attribuée sous forme d'un nombre entier en notation hexadécimale, après deux sauts de tabulation, sous forme des variables de structure nombre.moitie.octet1 et nombre.moitie.octet2.

Voici l'affichage produit par le programme UNION.C :

Valeur 2 octets	1er octet	2ème octet
64	64	0
c8	ffc8	0
12c	2c	1
190	ff90	1
1f4	fff4	1
258	58	2
2bc	ffbc	2
320	20	3
384	ff84	3
3e8	ffe8	3
44c	4c	4
4b0	ffb0	4
514	14	5
578	78	5

Vous constatez que certaines données de type *char* sont retournées avec les initiales "*ff*". Cela est dû au fait que le type de données *char* est introduit par un signe relatif.

Dans le chapitre 15.4, nous reviendrons sur l'utilisation des unions, et montrerons que cette structure de données peut être employée judicieusement pour améliorer l'efficacité des programmes.

13.8. Définir des types de données à l'aide de typedef

Avant d'aborder, dans la section qui suivra, la programmation des système de gestion de bases de données, nous allons vous montrer que, grâce à l'instruction *typedef*, des imbrications complexes de structures peuvent être ordonnées de façon claire. Le programme qui nous servira d'exemple est une variante du programme IMBRIQUE.C, légèrement modifié.

Listing 62 : Extrait du programme DEFTYPE.C

```
[1 ] /* Chapitre 13.8 */
[2 ] /* Nom du programme: DEFTYPE.C */
[3 ] /* Exemple de définition d'un nouveau type de données à l'aide de
        typedef; */
[4 ] /* Modification du programme IMBRIQUE.C  */
[5 ]
[6 ] main()
[7 ] {
[8 ]
[9 ]   typedef struct
[10]   {
[11]     char nom[25];
[12]     int age;
[13]     char statut; /* m = marié, c = célibataire */
[14]   } personne;    /* Il existe maintenant un type de données dont  */
[15]                  /* le nom est personne */
[16]
[17]   typedef struct
[18]   {
[19]     int servicenr;            /* numéro du service */
[20]     char position[25];        /* Grade dans l'entreprise */
[21]   } service;
[22]
[23]  typedef struct
[24]   {
[25]     service srv;
[26]     personne caracteristique;
[27]   } employe;
[28]
[29]   employe personnel[53];       /* 53 personnes de type employe */
[30]
[31]   employe directeur,autres;    /* Le directeur est aussi un employé */
[32] }
```

Observez d'abord les lignes 29 à 31. Des variables y sont définies, parmi celles-ci un tableau dont le type de données est *employe*. Ce type de données n'est pas inclus au langage de programmation C, et a par conséquent dû être défini dans le programme lui-même. Cette définition se trouve dans les lignes 23 à 27. Vous

voyez que *employe* est une structure composée des éléments *srv*, et *caracteristique*, qui eux-mêmes n'appartiennent pas non plus à un type de données appartenant au langage C.

Le terme déterminant est ici le mot clé *typedef*, que vous trouvez dans les lignes 9 et 17.

typedef permet au programmeur de définir de nouveaux types de données, qui pourront ensuite être utilisés comme s'ils appartenaient à l'ensemble des types de données inclus dans le langage C.

Ont finalement été définis les nouveaux types de données *personne*, *service* et *employe*, ainsi qu'une structure composée de ces types de données. Retenez surtout que dans la ligne 29, aucune variable n'est déclarée : ce sont des types de données qui ont été déclarés.

La logique de fonctionnement et l'affichage de DEFTYPE.C et IMBRIQUE.C sont les mêmes. La seule différence entre les deux programmes réside dans l'organisation de la structure des données. L'avantage qu'apporte le travail avec des types de données définis par le programmeur est dû au fait que même des imbrications complexes de structures peuvent être programmées clairement.

13.9. Ecrire et lire des structures

Ce que vous venez d'apprendre à propos des structures et des unions constitue un fondement de la construction de systèmes de gestion de base de données. Il s'agit là de systèmes de fichiers qui mémorisent des données dans une structure prédéfinie, pour faciliter leur traitement ultérieur : la recherche d'informations, l'impression de listes, l'enregistrement d'importantes quantités de données.

Dans une base de données, les structures remplies de données concrètes sont appelées "enregistrements de la base de données". Un enregistrement peut être composé d'éléments dont les types de données sont distincts. Un élément d'un enregistrement est appelé "champ". Dans les champs, les informations concrètes sont mémorisées sous forme de caractères numériques ou alphanumériques. Une base de données est ainsi construite hiérarchiquement. Elle se décompose en enregistrements dont la structure est fixe, eux-mêmes composés de champs, dont chacun est d'un type de données unique,

et chaque champ renferme des caractères, plus petites unités d'information que la base de données.

Dans les rubriques qui suivent, vous découvrirez des fonctions du langage C garantes d'une écriture et d'une lecture efficace des enregistrements. Elles permettront de rechercher des enregistrements quelconques, de les visualiser, et, le cas échéant, de modifier leur contenu. Pour vos propres applications, vous pourrez modifier ces fonctions, et les adapter à vos besoins. Dans les programmes de démonstration qui suivent, vous retrouverez toute la palette des connaissances que vous avez acquises jusqu'ici à propos du C.

13.9.1. Ecriture d'enregistrements dans un fichier

L'exemple qui suit, qui concerne un fichier de numéros de téléphone, montre comment des enregistrements peuvent être écrits dans une base de données.

Listing 63 : Le programme STRUCDAT.C

```
[1 ] /* Chapitre 13.9.1 */
[2 ]
[3 ] /* Nom du programme : STRUCDAT.C */
[4 ] /* Programme de démonstration - Ecriture d'enregistrements  */
[5 ] /* à l'aide de la fonction fwrite() */
[6 ]
[7 ] #include "stdio.h"
[8 ] void saisie(void);            /*Prototype - Saisie d'enregistrements */
[9 ] #define FICHIER "a:\\telephon.dat" /* Le fichier sera créé */
[10]                                   /* dans la racine de l'unité A: */
[11] #define ESC 27                    /* code ASCII de la touche ESCAPE */
[12]
[13] typedef struct
[14]   {
[15]   char nom[40];                /* il s'agit là de                */
[16]   char pnom[40];               /* trois champs de données        */
[17]   char tel[12];                /* de la structure enregistrement  */
[18]   } enregistrement;            /* le nom de la structure que nous */
[19]                                /* définissons est enregistrement  */
[20]
[21]   enregistrement telinfo;      /* telinfo est une variable globale */
[22]                                /* du type de données enregistrement */
[23]
[24] main()
[25] {
[26]   int i,
[27]   l = sizeof(enregistrement);     /* l mémorise la longueur de la */
[28]                                   /* structure enregistrement     */
[29]   char touche = ' ';          /* touche est initialisée par un blanc */
```

```
[30]   FILE *fpt;                /* pointeur dirigé sur un fichier */
[31]
[32]   fpt = fopen(FICHIER,"a"); /* Créer un fichier ou , s'il existe,  */
[33]                             /* l'ouvrir en écriture pour y ajouter */
[34]                             /* des enregistrements                 */
[35]
[36]   while(touche != ESC)      /* tant que ESC n'est pas actionnée */
[37]   {
[38]     saisie();                    /* Entrer l'enregistrement          */
[39]     fwrite(&telinfo,l,1,fpt);    /* Ecrire l'enregistrement          */
[40]                                  /* la fonction a besoin de          */
[41]                                  /* l'adresse de l'enregistrement    */
[42]                                  /* à écrire, de sa taille,          */
[43]                                  /* nombre des enregistrements à     */
[44]                                  /* écrire et du pointeur dirigé sur */
[45]                                  /* le fichier                       */
[46]     printf("\nTouche quelconque pour continuer. - [ESC] pour finir\n");
[47]     touche = getch();
[48]   }
[49] }
[50]
[51] void saisie(void)
[52] {
[53]   printf("Veuillez entrer votre nom          : -> ");
[54]   scanf("%s",telinfo.nom);
[55]   printf("Veuillez entrer votre prénom       : -> ");
[56]   scanf("%s",telinfo.pnom);
[57]   printf("Veuillez entrer votre numéro de téléphone  : -> ");
[58]   scanf("%s",telinfo.tel);
[59] }
```

Le programme commence par l'intégration du fichier d'en-tête STDIO.H, ainsi que la définition de deux constantes symboliques, dont l'une est ESC (code décimal ASCII 27), et dont l'autre est *fichier*, constante qui sera remplacée pendant l'exécution du programme par un véritable nom de fichier. Ensuite, dans les lignes 13 à 18, *typedef* assure la définition d'une structure, que nous avons appelée *enregistrement*, et enfin, nous déclarons une variable *telinfo* dont le type de données est *enregistrement*. Etant donné que cette variable est déclarée avant toute autre partie du programme, elle est de type *globale*.

Dans le programme principal sont déclarées deux variables entières, une variable de type *char*, ainsi qu'un pointeur dirigé sur un fichier. Parmi les variables, vous voyez en ligne 27 la variable entière *l*, initialisée avec la longueur de l'enregistrement. Jusqu'ici, il n'y a donc rien de vraiment nouveau. Il en va différemment de la ligne 39. L'instruction

```
fwrite(&telinfo, l, 1, fpt);
```

demande au système d'écrire une seule et unique fois les données mémorisées dans *telinfo*, dont la longueur est de *l* octet, dans le fichier sur lequel pointe le

pointeur *fpt*. Notez bien que cette fonction a besoin de l'adresse de l'enregistrement, qui lui est d'ailleurs transmise à l'aide de l'opérateur d'adressage &.

La boucle de saisie, répartie sur les lignes 36 à 48, sera répétée jusqu'à ce que la valeur de "*touche*" soit celle de la constante symbolique ESC. Il s'agit là de la valeur décimale 27, qui correspond au code ASCII de la touche <Esc>.

Introduisez une disquette dans l'unité A: et lancez le programme. Entrez cinq enregistrements. Ne soyez pas surpris si les enregistrements ne sont pas immédiatement écrits sur la disquette après chaque cycle de la boucle de saisie. Le système d'exploitation mémorise dans un premier temps les données à enregistrer dans un buffer, qui ne sera vidé que dès qu'il est plein, ou que vous quittez le programme.

Chargez maintenant le fichier A:TELEPHON.DAT dans l'éditeur de votre compilateur. Si vous travaillez sous Turbo C 2.0, un message vous est retourné :

```
Line too long - CR inserted. Press <Esc>
```

Ce texte signifie que les lignes du fichier à charger sont trop longues, et qu'un "retour chariot" a dû être inséré au texte. Appuyez sur la touche <ESC> : l'éditeur chargera le fichier. Vous voyez une ligne dans laquelle les espaces vides des champs d'un enregistrement sont remplis de "@".

L'éditeur de C++, chargent sans difficulté ce fichier de numéros de téléphone. Vous voyez plus particulièrement qu'aucun saut de ligne ne suit un enregistrement. Dans cet éditeur, les espaces vides des champs de données sont toutefois représentés par de véritables espaces.

Le fichier est par conséquent une image de la structure que nous avions écrite à l'aide de *fwrite()*. Nous pouvons en déduire que la fonction *fwrite()* n'enregistre pas seulement les données, mais également la structure d'un enregistrement. Dans la section qui suit, vous verrez que la fonction *fread()* permet, grâce à cette technique, une lecture facile des informations contenues dans un enregistrement.

13.9.2. Extraction d'enregistrements d'un fichier à l'aide de fread()

Le programme qui suit ouvre le fichier TELEPHON.DAT en lecture, et renvoie à l'écran le contenu de ce fichier à l'aide de la fonction *fread()*.

Listing 64 : Le programme LISFICH.C

```
[1 ] /* Chapitre 13.9.2 */
[2 ] /* Nom du programme : LISFICH.C */
[3 ] /* Exemple de lecture d'un fichier structuré écrit à l'aide de  */
[4 ] /* fwrite(), fondement de la conception de bases de données     */
[5 ]
[6 ] #include "stdio.h"
[7 ] #define FICHIER "a:\\telephon.dat"
[8 ]
[9 ] typedef struct
[10] {
[11]   char nom[40];
[12]   char pnom[40];
[13]   char tel[12];
[14] } enregistrement;
[15]
[16] main()
[17] {
[18]   FILE *fpt;
[19]   enregistrement telinfo;
[20]
[21]   if ((fpt = fopen(FICHIER,"r")) == NULL)  /* si fichier non ouvert */
[22]       printf("Fichier %s non trouvé",FICHIER);    /* Message */
[23]   else                                            /* sinon */
[24]   {
[25]     while (fread(&telinfo,sizeof(telinfo),1,fpt)) /* Lecture */
[26]     {                                   * jusqu'à la fin du fichier */
[27]       printf("Nom %s \t Prénom : %s\t Téléphone : %s\n",
[28]              telinfo.nom, telinfo.pnom, telinfo.tel);
[29]     }                                             /* Fin de WHILE */
[30]   }                                               /* Fin de ELSE */
[31]
[32]   fclose(fpt);    /* Refermer le fichier ouvert */
[33] }
```

La partie du programme qui contient les déclarations renferme les mêmes instructions que dans notre programme de démonstration précédent. Une seule différence : le fichier TELEPHON.DAT est ouvert en lecture, avec interception d'éventuelles erreurs survenant lors de l'ouverture.

L'instruction

```
if ((fpt = fopen(FICHIER, "r") == NULL)
```

qui figure en ligne 21 teste si l'ouverture du fichier a réussi. Dans ce but, le résultat de la fonction *fopen()* est d'abord retourné au pointeur de fichier *fpt*. Il est alors vérifié si la valeur qui vient d'être attribuée est NULL. Si le résultat de cette comparaison est vrai, cela signifie que le fichier à ouvrir n'a pas pu être localisé, et que le fichier n'a donc pas été ouvert. Par contre, si la valeur renvoyée est différente de NULL, le pointeur est maintenant dirigé sur le fichier qui vient d'être ouvert en lecture.

La logique de la construction *if..else* utilisée ici prévoit qu'en cas d'erreur de lecture (c'est-à-dire si le fichier n'a pas pu être ouvert) un message d'erreur approprié soit renvoyé à l'écran. Ce n'est que si l'ouverture du fichier a été couronnée de succès que sera parcourue une boucle d'extraction (lecture) dans la ramification *else*.

Etant donné que la fonction *fread()* utilisée en ligne 25 renvoie une valeur positive entière pour chaque processus de lecture réussi, nous pouvons programmer le processus de lecture (lignes 25 à 29) sous forme d'une boucle *while*. La condition d'exécution de la boucle est vraie aussi longtemps que la lecture est possible : c'est-à-dire jusqu'à la fin du fichier. Vous pouvez constater d'autre part que la syntaxe à laquelle obéit *fread()* est identique à celle de la fonction *fwrite()*. Dans les deux cas, il faut comme arguments l'adresse et la taille de l'enregistrement à lire. La lecture concerne à chaque fois un enregistrement, le pointeur de fichier étant dirigé sur le fichier qui doit être lu.

Lancez le programme. Après quelques instants, vous voyez le contenu de notre fichier de numéros de téléphone. Avant d'aborder, dans la section suivante, d'autres techniques relatives à la programmation d'une base de données, nous vous suggérons d'ajouter des enregistrements supplémentaires dans ce fichier, à l'aide du programme STRUCDAT.C. Parmi les informations que vous allez entrer, faites figurer les données suivantes :

```
Monnet      Michel    40-21-12-53
Kuno        Henri     42-70-44-25.
```

13.9.3. Recherche et affichage sélectif de données

L'utilisation principale des bases de données réside dans la recherche et le renvoi sélectifs d'informations. Grâce à notre nouveau programme de démonstration, CHERCHE.C, nous apprendrons comment programmer un tel procédé en C.

Le programme mémorise un nom entré au clavier, et recherche dans les champs telinfo.nom du fichier A:\TELEPHON.DAT, un élément qui correspond à ce critère de recherche. Dès qu'un tel élément est localisé, le programme renvoie l'enregistrement correspondant à l'écran.

Listing 65 : Le programme CHERCHE.C

```
[1 ] /* Chapitre 13.9.3 */
[2 ] /* Nom du programme : CHERCHE.C */
[3 ] /* Exemple de recherche dans un fichier structuré */
[4 ] #include "stdio.h"
[5 ] #include "string.h"
[6 ] #define FICHIER "a:\\telephon.dat"
[7 ]
[8 ] typedef struct
[9 ] {
[10]   char nom[40];
[11]   char pnom[40];
[12]   char tel[12];
[13] } enregistrement;
[14]
[15] main()
[16] {
[17]   int trouve=0,                    /* 0 = faux, rien trouvé */
[18]   s = sizeof(enregistrement);      /* longueur de l'enregistrement */
[19]   FILE *fpt;
[20]   char critere_recherche[25];
[21]
[22]   enregistrement telinfo;
[23]
[24]   printf ("Entrez votre nom : -> ");   /* Chargement du critère */
[25]   scanf("%s",critere_recherche);       /* de recherche          */
[26]
[27]   if ((fpt = fopen(FICHIER,"r"))==NULL)
[28]       printf ("Erreur lors de l'ouverture du fichier ");
[29]
[30]   while (fread(&telinfo,s,1,fpt))      /* jusqu'à la fin du fichier */
[31]   {
[32]     if (!(strcmp(critere_recherche,telinfo.nom)))
[33]       /* lorsque les deux critères sont identiques, la valeur de */
[34]       /* strcmp est égale à 0 !!! et l'expression est
             logiquement fausse  */
[35]     {
[36]        printf("Nom : %s Prénom : %s  Téléphone : %s\n",
```

```
[37]                 telinfo.nom,telinfo.pnom,telinfo.tel);
[38]         trouve = 1; /* Critère de recherche localisé */
[39]       }             /* Fin de IF */
[40]    }                /* Fin de WHILE */
[41]
[42]    if (!trouve)     /* si rien trouvé, alors */
[43]                     /* trouve est égal à 0 ! */
[44]        printf ("Nom non trouvé !!!!!!!!");
[45]
[46]    fclose(fpt);
[47] }
```

Au début du programme, vous trouvez les mêmes déclarations que dans les exemples précédents. Une nouvelle variable apparaît toutefois en ligne 17 : il s'agit de "*trouve*", à laquelle est affectée la valeur initiale 0. La valeur que prend cette variable indique si le critère d'extraction de la ligne 25, *critere_recherche*, a pu être localisé dans le fichier qui a été lu. Vous pouvez d'ailleurs le constater vous-même, en comparant la chaîne recherchée (dans notre cas, le nom) avec le contenu du champ correspondant (dans notre cas *telinfo.nom*) de l'enregistrement en cours de lecture. Dans le programme, cette comparaison est réalisée à l'aide de la fonction *strcmp()*. Si les deux chaînes sont identiques, le résultat renvoyé par la fonction est 0, ce qui équivaut à la valeur logique "faux". Par conséquent, la signification de la ligne 32

```
if (!(strcmp(critere_recherche, telinfo.nom)));
```

est : "vérifie si les chaînes alphanumériques mémorisées dans les variables *critere_recherche* et *telinfo.nom* sont identiques. Si oui, c'est-à-dire si la valeur retournée par la fonction *strcmp()* est 0 (sa négation étant donc vraie), l'enregistrement dont la lecture est en cours sera renvoyé". Ce processus se répète jusqu'à ce que la fin du fichier soit atteinte.

Lancez le programme, et entrez, en guise de critère de recherche, le nom

```
Monnet
```

Comme vous pouvez le constater, les informations concernant la personne dont le nom est Monnet apparaissent à l'écran. Faites un autre essai, en utilisant comme critère de recherche :

```
Kuno
```

puis en utilisant un nom qui n'est pas mentionné dans la base de données.

Le programme de démonstration qui suit vous montre le mouvement du pointeur dirigé sur les enregistrements à l'intérieur d'un fichier. Le pointeur d'enregistrement indique la position à partir de laquelle s'effectue l'accès en lecture ou en écriture dans un enregistrement.

Ce programme affiche, dans un premier temps, la taille de notre fichier de numéros de téléphone, exprimée en octets. Ensuite, il se rend sur un enregistrement quelconque, et le renvoie à l'écran.

Listing 66 : Le programme DEPLACE.C

```
[1 ] /* Chapitre 13.9.3 */
[2 ] /* Nom du programme : DEPLACE.C */
[3 ] /* Exemple de déplacement du pointeur de fichier dans */
[4 ] /* un fichier structuré */
[5 ]
[6 ] #include "stdio.h"
[7 ] #define FICHIER "a:\\telephon.dat"
[8 ] typedef struct
[9 ] {
[10]   char nom[40];      /* il s'agit là de                  */
[11]   char pnom[40];     /* trois champs de données          */
[12]   char tel[12];      /* de la structure enregistrement */
[13] } enregistrement;
[14] long taille(FILE *pointer);   /* Prototype. La fonction retourne */
[15]                               /* une valeur de type long */
[16]
[17] main()
[18] {
[19]   int nr;                /* Numéro de l'enregistrement        */
[20]   long l,                /* Champ d'aide au déplacement dans le fichier */
[21]   pos,                        /* Position de l'enregistrement      */
[22]   g,                          /* Taille du fichier en octets       */
[23]   s = sizeof(enregistrement);/* Taille de la structure            */
[24]   FILE *fpt;
[25]   enregistrement telinfo;
[26]
[27]   fpt = fopen(FICHIER,"r");
[28]
[29]   g = taille(fpt);        /* déterminer taille du fichier */
[30]
[31]   printf("La taille du fichier %s est de %ld octets\n\n",FICHIER,g);
[32]
[33]   printf("Entrer le numéro d'un enregistrement -> ");
[34]   scanf("%d",&nr);     /* interrogation de l'enregistrement */
[35]
[36]   if (!(nr == 1))
[37]       l = s * (nr-1);
[38]   else
[39]       l = nr;
[40]
[41]   if (l > (g - s))   /* Numéro de l'enregistrement est trop grand */
```

```
[42]    {
[43]      printf("L'enregistrement numéro %d n'existe pas ",nr);
[44]      printf("\nFin du programme .................");
[45]      exit(0);          /* D'où message d'erreur et sortie du programme */
[46]    }
[47]
[48]    fseek(fpt,l,SEEK_SET);     /* positionnement à l caractères   */
[49]                               /* après le début du fichier       */
[50]    pos = ftell(fpt);          /* Position du pointeur de fichier */
[51]
[52]    printf("\n\nLe pointeur de fichier se trouve à la position %ld",pos);
[53]
[54]    fread(&telinfo,s,1,fpt);   /* Lis l'enregistrement à la */
[55]                               /* position actuelle         */
[56]    printf("\n\nL'enregistrement numéro %d est %s\t %s\t %s\n",
[57]            nr, telinfo.nom, telinfo.pnom, telinfo.tel);
[58] }
[59]
[60] long taille(FILE *pointer)
[61] {
[62]    long longueur;
[63]
[64]    fseek(pointer, 0L, SEEK_END);     /* SEEK_END Fin du fichier  */
[65]    longueur = ftell(pointer);        /* nombre d'octets du début */
[66]                                      /* à la fin du fichier      */
[67]    return(longueur);
[68] }
```

Lorsqu'un programme doit agir de la sorte, il faut d'abord réfléchir au moye qui permet de détecter l'emplacement sur lequel est dirigé le pointeur de fichie Deux positions sont faciles à localiser. Lors de l'ouverture du fichier, le pointe se trouve sur l'origine du fichier. Lorsque tous les enregistrements ont été lu à l'aide d'une boucle *while*, par exemple parcourue jusqu'à ce que la marq de fin de fichier EOF soit atteinte, le pointeur de fichier se trouve sur l'extrém de ce fichier.

Trois constantes symboliques permettant de localiser respectiveme les positions extrêmes et la position actuelle du pointeur sont défin dans le fichier d'en-tête STDIO.H. Il s'agit de

SEEK_SET pour le début du fichier,
SEEK_CUR pour la position actuelle dans le fichier
SEEK_END pour la fin du fichier.

Ces réflexions préalables ayant été menées, observez le programme. Dans la partie des déclarations, vous voyez, en ligne 14, le prototype d'une fonction :

```
long taille(FILE *pointer);
```

Celui-ci retourne une valeur entière du type de données *long*, et qui s'attend à recevoir en guise de paramètre un pointeur du type FILE, la valeur calculée représentant la taille du fichier exprimée en octets. Dans *main()*, en ligne 29, cette fonction est utilisée, et le résultat qu'elle renvoie et transmis à la variable *g*. Avant d'analyser la suite du fonctionnement de *main()*, analysons d'abord la fonction *taille()*, dont la définition est répartie sur les lignes 60 à 68.

La logique de cette fonction est relativement simple. La taille d'un fichier n'est rien de plus que la distance comprise entre son origine et son extrémité, exprimée en octets. Il suffit donc de positionner le pointeur du fichier sur l'extrémité de ce fichier, et de détecter sa distance par rapport à l'origine. A cet effet, nous utilisons deux fonctions. La fonction *fseek()* (de l'anglais file seek) positionne toujours le pointeur relativement à une position prédéfinie. Il s'en suit que l'instruction

```
fseek(pointer,0L,SEEK_END);
```

signifie : "positionne le pointeur dans le fichier sur lequel est dirigé pointer, à 0 octet au-dessous de la position de SEEK_END."

Par conséquent, le pointeur sera dirigé sur l'extrémité du fichier, c'est-à-dire la position de SEEK_END. L'indication du nombre d'octets duquel le pointeur doit être déplacé vers le bas par rapport à la position indiquée est toujours une valeur de type *long*, spécifiée par le descripteur de champ *l*.

Venons-en à la deuxième fonction utilisée, en ligne 65 : *ftell()*. Cette fonction renvoie la position actuelle du pointeur dirigé sur le fichier, exprimée en octets. Cette valeur correspond à la distance entre l'origine du fichier et la position actuelle du pointeur. Par conséquent, la proposition

```
longueur = ftell(pointer);
```

renvoie dans la variable *longueur* la taille du fichier, puisque le pointeur se trouve à cet instant sur la fin du fichier.

Si vous analysez maintenant la suite du déroulement du programme principal, vous voyez que, dans la ligne 34, un numéro d'enregistrement est chargé par

l'intermédiaire de la fonction *scanf()*, celui-ci étant ensuite converti en une valeur *l* par l'intermédiaire de l'instruction

```
l = s * (nr - 1)
```

contenue dans une instruction *if..else*. Cette valeur correspond à l'origine de l'enregistrement à lire, exprimée en octets. Pour tous les enregistrements dont le numéro est supérieur à 1, cette position est systématiquement égale au

```
produit de la longueur d'un enregistrement par
(numéro de l'enregistrement - 1).
```

Pour le premier enregistrement, cette position est toujours 1.

Il se pourrait toutefois que l'utilisateur entre un numéro d'enregistrement qui soit trop grand. La fonction calculerait alors une position qui dépasserait l'extrémité réelle du fichier. Une telle erreur est interceptée dans la ligne 41 à l'aide de

```
if (l < (g - s) )
```

Cette comparaison logique est vraie lorsque la spécification de position *l* est supérieure à la dernière position possible d'un enregistrement. Celle-ci résulte de la différence entre la longueur du fichier, mémorisée dans la variable *g*, et la longueur d'un enregistrement mémorisée dans la variable *s*.

Lorsque la position de l'origine de l'enregistrement à retourner a été déterminée, et que d'éventuelles erreurs de saisie ont pu être exclues, la fonction *fseek()* dirige le pointeur sur l'enregistrement à lire, par l'intermédiaire de

```
fseek(fpt, l, SEEK_SET);
```

qui se trouve en ligne 48, et qui place le pointeur à *l* caractère derrière le début du fichier. Cette fois, le positionnement est réalisé relativement à l'origine du fichier symbolisée par SEEK_SET. Avant que l'enregistrement sur lequel a été positionné le pointeur ne soit lu (ligne 54) puis renvoyé (ligne 56), la fonction *ftell()* détermine la position courante du pointeur, la transmet à la variable *pos*, le contenu de cette dernière étant ensuite retourné à l'écran.

Nous vous conseillons maintenant de "jouer" avec ce programme, et de vérifier s'il produit effectivement les résultats attendus.

13.9.4. Modifier des enregistrements

En raison des performances que nous en attendions, le programme de démonstration précédent est relativement volumineux. Un certain temps est nécessaire pour saisir sa structure. Dans la présente section, vous verrez que toutefois des programmes volumineux peuvent aussi être rédigés de façon plus structurée, en évitant les répétitions de déclarations et de définitions.

Le programme qui suit est le premier qui met en évidence l'un des points fort du langage de programmation C. Nous entendons par là la possibilité de développer des programmes modulaires. Un programme modulaire est un programme composé de plusieurs blocs, indépendants les uns des autres.

Listing 67 : Le programme MODIFIE.C

```
[1 ] /* Chapitre 13 */
[2 ] /* Nom du programme : MODIFIE.C */
[3 ] /* Exemple de modifications d'enregistrements et de */
[4 ] /* Programmation modulaire */
[5 ]
[6 ]
[7 ] #include "telephon.c"/* Intégration de fonctions, structures de */
[8 ]                       /* données et macros définies par le programmeur*/
[9 ] main()
[10] {
[11]   if (init(FICHIER)) /* si erreur lors de l'ouverture du fichier */
[12]       exit(1);       /* alors quitter le programme */
[13]   if (charger())
[14]       exit(1);      /* Erreur - Numéro d'enregistrement erroné - Fin */
[15]   editer();         /* Charger et éditer champ de données */
[16]   enregistrer();    /* Sauvegarde */
[17] }                   /* et fin */
```

Vous voyez que ce programme de démonstration ne renferme aucune déclaration, et qu'il est composé d'un petit nombre de lignes. Vous y trouvez juste l'activation des fonctions *init()*, *charger()*, *editer()*, et *enregistrer()*. Ces fonctions sont programmées dans le fichier d'inclusion TELEPHON.C, qui est intégré au programme dans la ligne 7. Dans le programme principal de cet exemple, le travail est donc exclusivement exécuté à l'aide de fonctions programmées dans un fichier indépendant de celui dans lequel nous nous trouvons. Vous pouvez donc faire comme si ces fonctions étaient des éléments du langage de programmation.

Lorsque vous lancez MODIFIE.C, vous constatez que, même si ce programme est bien plus petit que nos exemples précédents, il permet d'éditer (c'est-à-dire modifier) des enregistrements quelconques du fichier TELEPHON.DAT. Ce processus complexe nécessite les actes suivants :

① Il faut d'abord ouvrir le fichier.

② Il faut ensuite sélectionner un enregistrement. Cette sélection suppose qu'il est possible d'entrer le critère d'extraction au clavier.

③ L'enregistrement sélectionné doit ensuite être chargé.

④ L'enregistrement doit être modifié puis

④ Sauvegardé.

La collection de fonctions TELEPHON.C met à notre disposition toutes les fonctions décrites ci-dessus.

Listing 68 : Le fichier d'inclusion TELEPHON.C

```
[1 ] /* TELEPHON.C                                                    */
[2 ] /* Routines                                                      */
[3 ] /* Structures de données                                         */
[4 ] /* Constantes              pour le programme MODIFIE.C           */
[5 ] /*                                                               */
[6 ] /* dernière modification : 02.12.1992                            */
[7 ] /*
[8 ]
[9 ] /* Fichiers d'inclusion nécessaires */
[10] #include <stdio.h>
[11]
[12] /* Prototypes */
[13] long taille(FILE *fp);/* Prototype de la fonction taille, qui      */
[14]                       /* retourne une valeur de type long, et      */
[15]                       /* détermine la taille de fichiers quelconques */
[16]
[17] int init(char *fichier);       /* Ouverture de fichiers quelconques */
[18]
[19] int charger(void);   /* Lire et vérifier numéro de l'enregistrement */
[20]
[21] void positionner(int start); /* Pointeur d'enregistrement sur start */
[22]
[23] void editer(void);                /* Editer l'enregistrement courant */
[24]
[25] int enregistrer(void);      /* Enregistrer l'enregistrement courant */
[26]
[27]
```

```
[28] /* Constantes et macros */
[29] #define FICHIER "a:\\telephone.dat"
[30]
[31]
[32] /* Structure de données */
[33] typedef struct
[34] {
[35]   char nom[40];        /* il s'agit là de                  */
[36]   char pnom[40];       /* trois champs de  données         */
[37]   char tel[12];        /* de la structure enregistrement */
[38] } enregistrement;
[39]
[40]
[41] /* variables globales */
[42] enregistrement telinfo;
[43]
[44] int nr;                  /* Numéro de l'enregistrement  */
[45] long l,                  /* Champ d'aide le pointeur d'enregistrement */
[46] pos,                     /* Position de l'enregistrement */
[47] g,                       /* Taille du fichier en octets */
[48] s = sizeof(enregistrement); /* Taille de la structure enregistrement */
[49] FILE *fpt;               /* pointeur dirigé sur un fichier */
[50]
[51]
[52] /* Bibliothèque de fonctions */
[53]
[54] int init(char *fichier)      /* Ouverture d'un fichier quelconque  */
[55] {
[56]   if ( (fpt = fopen(fichier,"r+")) == NULL) /* en lecture */
[57]                                             /* et en écriture */
[58]   {
[59]      printf("Impossible d'ouvrir le fichier %s \n",fichier);
[60]      printf("Fin du programme.........................");
[61]      return(1);
[62]   }
[63]
[64]   g = taille(fpt);    /* Déterminer la taille du fichier */
[65]   printf("La taille du fichier %s est de %ld octets\n\n",FICHIER,g);
[66]   return(0);
[67] }
[68]
[69] int charger(void)    /* Chargement d'enregistrement -
                             Routine de détection d'erreur */
[70] {
[71]   printf("Entrez le numéro de l'enregistrement -> ");
[72]   scanf("%d",&nr);    /* Interrogation de l'enregistrement */
[73]
[74]   if (!(nr == 1))
[75]       l = s * (nr-1);
[76]   else
[77]       l = nr;
[78]
[79]   if (l > (g - s))  /* Le numéro de l'enregistrement est trop grand */
[80]   {
[81]      printf("L'enregistrement numéro %d n'existe pas ",nr);
```

```
[82]      printf("\nFin du programme .................");
[83]      return(1);       /* D'où message d'erreur et sortie du programme */
[84]    }
[85]    return(0);
[86] }
[87]
[88] void positionner(int start)       /* Positionnement du pointeur
                                 d'enregistrement à un endroit quelconque */
[89] {
[90]    fseek(fpt,start,SEEK_SET);     /* Positionne à l caractères */
[91]                                   /* après le début du fichier */
[92]    pos = ftell(fpt); /* Détermine la position du pointeur de fichier */
[93]
[94]    printf("\n\n Le pointeur du fichier se trouve à la position
                 %ld",pos);            /* Renvoi de la position */
[95]
[96]   }
[97]   void editer(void)      /* Positionner, charger et éditer */
[98] {
[99]
[100]    positionner(l);
[101]    fread(&telinfo,s,1,fpt);     /* Lis l'enregistrement à la */
[102]                                 /* position actuelle         */
[103]    printf("\n\nAge Nom : %s\n",telinfo.nom);
[104]    printf("\n\nVeuillez entrer le nouveau nom -> ");
[105]    scanf("%s",telinfo.nom);
[106] }
[107]
[108] int enregistrer(void)           /* Positionner et enregistrer */
[109] {
[110]    positionner(l);
[111]    fwrite(&telinfo,s,1,fpt);
[112] }
[113]
[114] long taille(FILE *fp)          /* Déterminer la longueur du fichier */
[115]    {
[116]    long longueur;
[117]
[118]    fseek(fp, OL, SEEK_END);     /* SEEK_END est la fin de fichier */
[119]    longueur = ftell(fp);        /* nombre d'octets du début à     */
[120]                                 /* la fin du fichier              */
[121]    return(longueur);
[122] }
```

Dans le listing ci-dessus, vous trouverez toutes les structures de données, constantes et fonctions utilisées dans le programme MODIFIE.C. Les structures de données vous sont connues depuis nos exemples antérieurs. Vous pouvez donc entièrement vous concentrer sur les fonctions.

La fonction *init()* répartie sur les lignes 54 à 67 ouvre un fichier dont le nom lui est communiqué par l'intermédiaire d'un pointeur de type *char*. Lorsque ce fichier ne peut être ouvert (c'est-à-dire que *fopen()* renvoie NULL), un message

d'erreur est renvoyé, et la valeur 1 est retournée à la fonction appelante. De plus, la fonction *taille()* détermine le volume du fichier, en octets, cette information étant requise par la suite pour positionner le pointeur sur le fichier.

La fonction *charger()*, définie dans les lignes 69 à 86, lit le critère de sélection (ici le numéro de l'enregistrement), et vérifie si la valeur saisie n'est pas erronée. La seule erreur possible est liée à la saisie d'un numéro d'enregistrement trop grand. C'est exactement cette éventualité qui est vérifiée dans les lignes 74 à 84. A cet effet, le programme calcule d'abord la position du pointeur d'enregistrement et la mémorise dans la variable *l*. La ligne 75

```
l = s * (nr - 1);
```

qui n'est traitée que si le numéro d'enregistrement saisi est supérieur à 1, détermine la position en fonction de la longueur de l'enregistrement, c'est-à-dire la valeur de

```
s = sizeof(enregistrement)
```

multipliée par (numéro d'enregistrement saisi - 1). Cette position correspond alors exactement à l'emplacement auquel l'enregistrement dont le numéro a été saisi se trouve dans le fichier. La comparaison

```
l > (g-s)
```

qui suit est toujours vraie lorsque la valeur calculée de *l* est trop grande. Dans ce cas, un message d'erreur sera affiché, et la valeur 0 sera retournée à la fonction appelante. La fonction *charger()* est donc "fausse" (puisqu'elle produit la valeur 0), dès qu'un numéro d'enregistrement trop grand a été entré.

Les fonctions *positionner()*, *editer()* et *enregistrer()* ne sont que les répétitions de structures que nous connaissons déjà. Notez toutefois qu'il est important, avant une nouvelle sauvegarde de l'enregistrement à modifier, de repositionner le pointeur dirigé sur le fichier à l'emplacement où il avait été positionné pour la lecture de l'enregistrement. C'est la raison pour laquelle, dans les fonctions editer() et *enregistrer()*, dans les lignes 100 et 108, est d'abord activée la fonction *positionner()* avec la même valeur.

Les éléments du programme enregistrés dans TELEPHON.C peuvent être considérés comme des entités indépendantes pouvant être intégrés dans tout programme à l'aide de l'instruction include. De telles entités sont également appelées modules. Le programme que nous avons présenté dans cette section

est un premier exemple de ce que l'on appelle la programmation modulaire. Les modules peuvent être construits indépendamment du programme principal, et communiquent les valeurs qu'ils retournent, par l'intermédiaire de paramètres, aux parties de programme appelantes, qui elles-mêmes peuvent aussi être des modules. Si l'on prend la précaution de bien séparer les variables globales et locales, il est possible de programmer les modules d'une façon suffisamment générale pour qu'ils puissent être utilisés dans les programmes les plus divers.

Un projet de "gros" programme n'est pas imaginable sans programmation modulaire. Le fait que le langage C supporte pleinement ce principe constitue l'un de ses principaux points forts.

13.10. Récapitulation

Les structures sont des compositions de types de données. Les éléments qui composent une structure peuvent être des types de données simples, ou des structures préalablement définies. Les structures peuvent donc être imbriquées. Leur définition s'effectue par l'intermédiaire des mots clé *struct* et *typedef*. Une structure nommée porte un nom qui suit immédiatement le mot clé *struct*.

Typedef permet de définir de nouveaux types de données, qui peuvent ensuite être utilisés dans le programme comme s'ils étaient implémentés dans le langage C.

L'accès aux divers champs d'une structure est réalisé par chaînage des noms des champs de cette structure. Ceux-ci doivent alors être séparés par un point. Le symbole -> permet d'accéder à un champ d'une structure à l'aide d'un pointeur dirigé sur cette structure. Ainsi,

```
pointeur -> identification
```

et

```
(*pointeur).identification
```

ainsi que

```
employe[i].identification
```

concernent le même champ de variable lorsque "*pointeur*" est un pointeur dirigé sur le tableau de la structure *employe[]*.

La définition d'unions permet de mémoriser à un même emplacement de la mémoire des données de types différents. Cela équivaut à accéder à une même adresse de la mémoire au moyen de noms de variables distincts. Cette technique permet de construire des structures de base de données complexes.

Les structures sont le fondement de la programmation de système de gestion de base de données. *fwrite()* permet d'écrire des fichiers structurés. Ces fichiers renferment des enregistrements dont l'organisation est l'image de la structure définie. Les éléments de la structure d'un enregistrement sont appelés champs. Ceux-ci peuvent renfermer des données de types distincts. La lecture des enregistrements est réalisée au moyen de la fonction *fread()*. La fonction *ftell()* renvoie, en guise de résultat, la position courante du pointeur du fichier, alors que la fonction *fseek()* positionne ce pointeur. Les constantes symboliques SEEK_SET, SEEK_CUR et SEEK_END représentent respectivement l'origine du fichier, la position courante du pointeur, et l'extrémité du fichier.

Les fonctions, structures de données et constantes nécessaires dans des programmes complexes peuvent être programmées dans des fichiers Include indépendants du programme proprement dit, puis intégrées dans celui-ci au moyen de l'instruction *#include*. Le programme lui-même reste alors compact et facile à analyser. Il ne se compose plus que d'éléments préprogrammés, appelés modules.

Exercices se rapportant au chapitre 13

Exercice 1 :
Ecrivez un programme qui, dans une structure, charge les éléments d'une adresse : nom, rue, ville et numéro de téléphone, et qui les renvoie à l'écran. Pour les éléments qui composent une adresse, vous déclarerez une structure globale que vous appellerez enregistrement.

Pour la saisie, vous programmerez une fonction

```
input_enregistrement(void)
```

Cette fonction devra charger des enregistrements jusqu'à ce que la saisie du nom soit conclue par <x> <Return>. Pour le chargement des champs des enregistrements, vous utiliserez la fonction

```
saisie()
```

décrite dans la section 11.5, dans le programme de démonstration INPUT.C.

Exercice 2 :
Développez le programme précédent de manière à ce que les enregistrements soient écrits dans un fichier appelé C:\TC\ADR.DAT. Le programme devra proposer à l'utilisateur le choix entre la saisie d'enregistrements, l'affichage d'enregistrements, et une option permettant de quitter le programme. Vous déclarerez la constante FICHIER pour le nom du fichier, et un pointeur global *fp* dirigé sur un fichier.

Vous programmerez aussi les fonctions suivantes :

```
int init_fichier(char *nomfichier)
```

pour ouvrir le fichier. Cette fonction devra renvoyer la valeur 0 lorsque le fichier n'a pas pu être ouvert.

```
int affichage(FILE *pointeurfichier)
```

qui devra pouvoir visualiser dans un format agréable à l'oeil le contenu du fichier d'adresses. Elle retournera la valeur 0 dès qu'une erreur est détectée.

```
void fin_prog(void)
```

qui referme le fichier ouvert, renvoie un message approprié à l'écran, et quitte le programme.

Vous construirez le programme principal de la manière suivante, et utiliserez à cet effet une version modifiée de la fonction get_keyboard qui avait été programmée dans l'exercice 3 du chapitre 11 :

```
main()
{
   char c;
   init_fichier(FICHIER);
   while ( xc = get_keyboard())
        {
        switch(c)
        {
             case '1': input_enregistrement();break;
             case '2': affichage(fp);break;
             case '3': fin_prog();
        }
   }
}
```

14

Gestion dynamique de la mémoire et traitement de listes

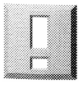
Le sujet principal de ce chapitre concerne la gestion dynamique de la mémoire. Vous y apprendrez à construire des listes à simple et double liaison, à insérer, modifier et effacer des éléments.

Pour le néophyte, la gestion dynamique de la mémoire est une notion qui peut ne pas sembler évidente. Au fur et à mesure de votre progression dans ce chapitre, vous verrez toutefois que cette technique de programmation n'est pas vraiment difficile, et qu'elle est souvent utile.

Jusqu'ici, vous n'avez travaillé en connaissance de cause qu'avec des constantes statiques. Les variables dynamiques, que nous avions appelées variables automatiques lorsque nous travaillions sur le domaine de définition des variables (chapitre 6.5) étaient générées par le système. Dans les paragraphes qui suivent, vous apprendrez à créer vous-même des variables dynamiques, puis à les effacer.

Les variables dynamiques sont des variables qui ne sont créées et pour lesquelles n'est réservée une plage de mémoire que lorsque leur usage s'impose.

Vous pouvez par conséquent créer autant de variables que vous le souhaitez dès lors que vous en avez besoin, et libérer l'espace mémoire qu'elles occupent lorsqu'elles ne sont plus utiles.



Une variable est donc qualifiée de "dynamique" dès lors qu'elle est créée puis effacée pendant l'exécution du programme.

Comme d'habitude, nous entamerons notre étude avec un exemple. Le programme qui suit relie des enregistrements entre eux, pour lesquels une plage de mémoire ne sera réservée que pendant le déroulement du programme. Une fois que les données auront été renvoyées à l'écran, l'espace mémoire qu'elles occupaient sera à nouveau libéré.

Listing 69 : Le programme CHAINE.C

```
[1 ] /* Chapitre 14.7 */
[2 ]
[3 ] /* Nom du programme : CHAINE.C */
[4 ] /* Exemple de liste à chaînage dynamique simple */
[5 ]
[6 ]
[7 ]
[8 ] #include "stdio.h"                    /* définition de NULL  */
[9 ] #define ENREGISTREMENTS 6
[10]
[11] main()
[12] {
[13]   struct enregistrement
[14]   {
[15]       char nom[25];
[16]       char pnom[25];
[17]       int age;
[19]       struct enregistrement *next;  /* Pointeur dirigé sur
                                            l'enregistrement suivant */
[20]   } *pointeur_courant, *start, *fin;/* Ce ne sont pas des variables! */
[21]
[22]   int index;
[23]
[24]   /* le premier enregistrement est toujours un cas particulier */
[25]
[26]   start=(struct enregistrement *)malloc(sizeof(struct enregistrement));
[27]   strcpy(start->nom,"Durand");    /* Durand est le premier     */
[28]   strcpy(start->pnom,"Jean");     /* Son prénom est Jean       */
[28]   start->age = 20;                /* et il a 20 ans            */
[29]   start->next = NULL;             /* start pointe sur NULL     */
[30]   fin = start;      /* fin est aligné sur l'élément de départ */
[31]
[32]   /* nous utilisons une boucle pour remplir */
[33]   /* les enregistrements restants           */
[34]   for (index = 0;index < ENREGISTREMENTS;index++)
[35]   {
[36]     pointeur_courant = (struct enregistrement*) malloc
                            (sizeof(struct enregistrement));
[37]     strcpy(pointeur_courant->nom,"Durand");
[38]     strcpy(pointeur_courant->pnom,"Michel");
[39]     pointeur_courant->age = 3+index;
[40]     fin->next = pointeur_courant;     /* dernier enregistrement */
[41]     pointeur_courant->next=NULL;      /* pointe sur NULL        */
[42]     fin = pointeur_courant;  /* Ceci est l'enregistrement précédent */
[43]   }
```

```
[44]
[45]   /* A partir d'ici, on imprime ! */
[46]
[47]   pointeur_courant=start;          /* Premier élément de la liste */
[48]   do
[49]   {
[50]     fin = pointeur_courant->next; /* fin pointe sur l'élément suivant*/
[51]
[52]     printf("%s est un %s, et a %d ans.\n",
[53]            pointeur_courant->nom, pointeur_courant->pnom,
[54]            pointeur_courant->age);
[55]     /* Imprime l'élément sur lequel est dirigé pointeur_courant */
[56]
[57]      pointeur_courant = pointeur_courant->next;
[58]     /* le pointeur actuel pointe sur l'élément suivant de la liste */
[59]   } while (fin != NULL);
[60]     /* jusqu'à ce que fin soit dirigé sur la fin de la liste */
[61]
[62]     /* Libérons maintenant l'espace mémoire occupé */
[63]
[64]   pointeur_courant = start;          /* premier bloc de données */
[65]   do
[66]   {
[67]       fin = pointeur_courant->next;  /* bloc de données suivant */
[68]       free(pointeur_courant);        /* Libérer le bloc */
[69]       pointeur_courant = fin;        /* Bloc suivant */
[70]   }while (fin != NULL);              /* jusqu'à ce que la fin de
                                             liste soit atteinte */
[71] }
```

Dans ce programme est déclarée une structure nommée dont les éléments définissent les caractéristiques d'une personne. Le nom de la structure est *enregistrement*, et trois pointeurs lui sont affectés. Vous n'avez donc déclaré aucune variable (propriété déterminante pour comprendre les réflexions qui vont suivre), et plus particulièrement aucun élément dans lequel pourraient être mémorisées des valeurs ! Nous ne disposons en fait que de trois pointeurs dirigés sur des variables dont la structure est *enregistrement*. Pour que ce programme puisse travailler, il vous faut générer des variables pendant l'exécution du programme. Il s'agira donc de variables dynamiques.

14.1. Créer des variables dynamiques sur le tas

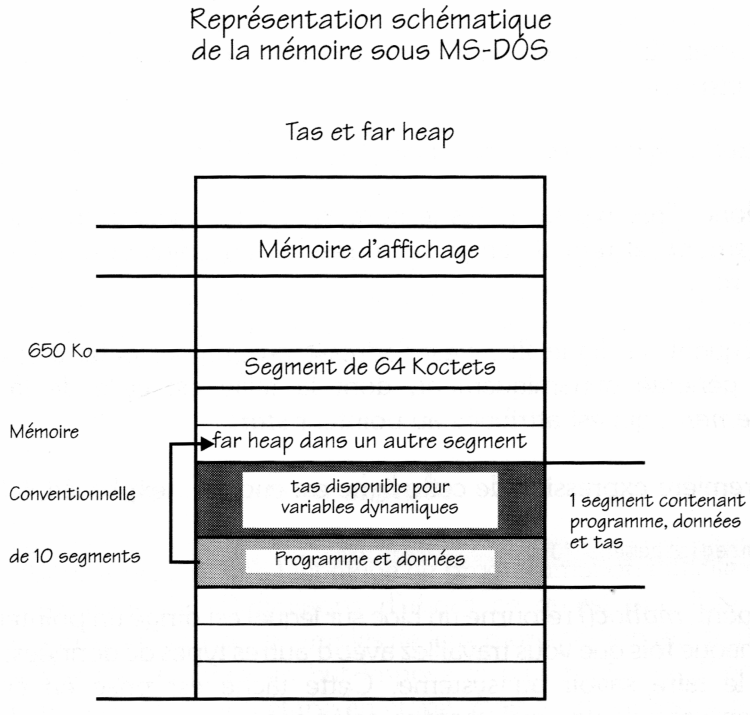
L'instruction contenue dans la ligne 26 génère une structure dynamique composée des trois variables de l'enregistrement. Par la même occasion, l'adresse de l'origine du bloc de mémoire réservé par la fonction *malloc()* est attribuée au pointeur *start*. La taille de ce bloc de mémoire est exactement celle de la structure de données *enregistrement*, et ce bloc occupe une zone de la mémoire de travail qui n'est allouée ni au programme, ni à des variables statiques, ni à des constantes. Cette zone de mémoire est appelée *tas*. Avant d'approfondir la fonction *malloc()*, nous allons voir plus précisément ce qu'est le *tas* (*heap* en anglais).

Le système d'exploitation MS-DOS n'est capable d'exploiter que 640 Koctets de mémoire de travail, au maximum. La gestion de cette plage de mémoire, également appelée mémoire conventionnelle, est assurée après décomposition en un maximum de 10 segments de 64 Koctets chacun.

Le tas est la partie d'un segment qui n'est pas occupée par le programme lui-même, ni par les constantes ou variables statiques intégrées au programme.

Sur le tas, il est possible de créer des variables dynamiques en cas de besoin, puis de les effacer. La gestion de cette zone est assurée par le système d'exploitation : il suffit donc au programmeur d'utiliser les fonctions du C prévues à cet effet.

Si, pour créer une variable dynamique, vous désirez accéder à une zone de mémoire qui n'appartient pas au même segment que celui occupé par le programme lui-même, la plage concernée est appelée *far heap*. Toutes les fonctions de gestion du tas sont applicables au *far heap*. Dans les sections qui suivent, vous vous plongerez dans l'étude de la gestion dynamique de la mémoire. Nous nous baserons sur l'exemple d'une gestion de liste pour développer une méthode qui permet de travailler dans un même programme avec un nombre illimité de variables.



Organisation de la mémoire sous MS-DOS

14.2. Allocation dynamique de la mémoire à l'aide de malloc()

La fonction *malloc()* demande au système de mettre à disposition un bloc de mémoire dont la taille est imposée par l'argument spécifié dans les parenthèses de la fonction. Le résultat de la fonction est alors un pointeur dirigé sur le début de ce bloc, et mémorisé sur le tas.

Il ne serait pas judicieux que le programmeur ait à déterminer lui-même le nombre d'octets nécessaires à un certain type de données ou à une structure définie. Cette tâche est prise en charge par la fonction *sizeof()*, que nous avons vue dans le chapitre 13.9, et que nous allons maintenant approfondir. Le résultat retourné par *sizeof()* est la valeur de l'argument nécessaire à *malloc()*, exprimée en octets. Ainsi, le résultat produit par

```
sizeof(struct enregistrement)
```

est de 52 octets (2*25 octets pour les deux chaînes, et 1*2 octets pour le nombre entier). L'instruction

```
malloc(sizeof(struct enregistrement))
```

signifie alors : "réserve sur le tas le nombre d'octets nécessaire à la structure *enregistrement*, et renvoie en guise de résultat un pointeur dirigé sur ce bloc de mémoire".

Par conséquent, en ligne 26, c'est l'adresse du premier octet d'une zone de mémoire générée dynamiquement, dont la taille est celle de la structure *enregistrement*, qui est attribuée au pointeur *start*.

Mais la première expression de cette ligne est encore inexpliquée :

```
(struct enregistrement *)
```

Implicitement, *malloc()* retourne un bloc sur lequel est dirigé un pointeur de type *char*. A chaque fois que vous travaillez avec d'autres types de données, il faudrait toutefois le faire savoir au système. Cette tâche est prise en charge par l'expression précédente, également appelée *cast*.

Le *cast* est une opération qui modifie le type de données des arguments qui suivent.

Le type de données vers lequel la conversion doit être effectuée est écrit dans les parenthèses de la fonction. Dans notre cas, nous demandons au système, par l'intermédiaire du cast, de convertir le pointeur de type *char* livré par la fonction *malloc()* en un pointeur du type *struct enregistrement*.

Lorsque le *cast* est absent, de nombreux compilateurs renvoient un message d'erreur, mais produisent malgré tout des valeurs correctes. Ce n'est pas une raison pour vous dispenser de cet opérateur : il garantira le bon fonctionnement de votre programme sur d'autres systèmes, et avec d'autres compilateurs.

A présent, vous disposez d'un espace mémoire approprié pour la structure, et des données peuvent être saisies. Comme vous le savez depuis les chapitres consacrés aux structures et aux pointeurs, il est possible d'accéder à tout élément d'une structure à l'aide d'un pointeur. Les lignes suivantes montrent une nouvelle fois le principe de l'affectation de données à des éléments d'une structure à l'aide

d'un pointeur. Les lignes 27 à 29 vous prouvent par la même occasion qu'il n'y a aucune différence avec l'utilisation de variables à définition statique.

Dans la suite du programme, en ligne 30, la valeur de *start* est affectée au pointeur *fin* : les deux pointeurs sont dès lors dirigés sur la même variable mémorisée sur le tas. Le pointeur *start* est dirigé sur la première structure dynamique créée, et le pointeur *fin* sur la dernière. Etant donné que, jusqu'à présent, seul un unique bloc de données a été créé, les deux pointeurs sont dirigés sur le même bloc de données. Dans la boucle *for* programmée dans les lignes 34 à 43, nous pouvons maintenant voir comment est générée à l'aide d'un troisième pointeur, *pointeur_courant*, une liste dans laquelle chaque élément possède l'adresse de l'élément suivant.

14.3. Listes à liaison simple

Nous abordons à présent une technique de programmation difficile concernant la notion de liste liée. Définissons d'abord ce qu'est une liste liée. Vous connaissez certainement des exemples de jeux d'orientation (ou de pistes), dans lesquels il s'agit d'atteindre un but imposé le plus rapidement possible. Les joueurs doivent progresser d'étape en étape, mais n'ont à chaque fois connaissance que d'une étape unique. A chaque étape ils découvrent un indice permettant d'atteindre l'étape suivante, jusqu'à ce que soit atteinte la dernière étape : la destination finale. Ces différentes étapes peuvent être assimilées à une liste liée.

Notre exemple utilise la même technique, tout en exploitant l'extrême rapidité de l'ordinateur. Voyons comment cette technique est mise en "pratique informatique".

14.3.1. Construction

Dans la structure *enregistrement* définie dans ce programme de démonstration, le quatrième élément est nouveau : il s'agit d'un pointeur. Celui-ci est dirigé sur une autre structure, de même type(!), et sera utilisé pour pointer sur l'élément suivant dans la liste, qui lui-même contient un pointeur dirigé sur l'élément qui le suit, et ainsi de suite. De cette conception il résulte une liste liée dans laquelle le pointeur est le maillon qui assure la liaison avec l'élément suivant de la liste.

Dans la ligne 20 sont déclarés trois pointeurs dirigés sur la structure *enregistrement*. Vous disposez donc d'éléments en quantité suffisante pour créer une liste, et la remplir de données.

14.3.2. Le premier élément d'une liste

Le premier champ d'une liste est un cas particulier, et son traitement est différent de celui de ceux qui suivent. Dans un premier temps, un espace mémoire approprié est réservé à l'aide de la fonction *malloc()* que vous connaissez, et les champs de la structure sont remplie de données. Par la même occasion, la constante symbolique NULL est attribuée au pointeur *start->next* (ligne 29). Son rôle est de signaler que la fin de la liste est atteinte, le pointeur *next* n'étant dirigé sur aucun autre élément suivant de la liste.

Vous dirigerez toujours le pointeur *start* sur le premier élément de la liste. Chaque traitement de la liste commence donc par l'élément sur lequel est dirigé le pointeur *start*. Ensuite, dans le programme, vous dirigerez aussi le pointeur *fin* sur la première structure. Pourquoi ? Nous n'allons pas tarder à le découvrir.

Retenez surtout dans l'immédiat que :

Le premier et le dernier élément d'une liste doivent être considérés différemment des autres éléments de cette liste. Il est impératif que le premier élément d'une liste, son origine, soit connu. Il faut aussi que le dernier élément pointe sur la fin de la liste.

14.3.3. Les autres éléments d'une liste

Le groupe d'instructions suivant, réparti sur les lignes 34 à 43, est constitué d'une boucle *for* à l'aide de laquelle sont générés autant d'éléments de la liste que le spécifie la constante ENREGISTREMENTS. Ces instructions utilisent à cet effet le pointeur *pointeur_courant* pour réserver des blocs de mémoire, et remplissent de valeurs les trois premiers champs de la structure. Ensuite, l'adresse du nouveau bloc de données est affectée au pointeur de l'élément précédent de la liste (il s'agit de *fin.next*), soit *pointeur_courant*. A présent, le bloc de données précédent dispose de l'adresse du bloc suivant, qui est le bloc de données courant. Le pointeur *pointeur_courant* est donc toujours dirigé sur l'élément de la liste sur lequel vous travaillez. Lorsque, pour finir, nous affectons la valeur NULL au pointeur *next* du bloc de données actuel, nous le désignons

comme étant le dernier élément de la liste. Celui-ci est, pour l'instant, suivi de NULL, ce qui équivaut à "pas d'autre adresse". L'affectation

```
fin = pointeur_courant;
```

attribue au pointeur *fin* l'adresse de l'élément courant de la liste qui, pour l'instant, est le dernier. A présent, le pointeur *pointeur_courant* est à nouveau disponible pour la création d'autres éléments de la liste.

A la fin de ces affectations, le pointeur *start* est toujours dirigé sur le premier élément, et le pointeur *fin* sur l'extrémité de la liste, soit NULL. Chaque élément de la liste, à l'exception du dernier, possède l'adresse de l'élément qui le suit directement. La liste est terminée. Tout cela peut paraître déconcertant, mais est tout à fait sensé ! Nous vous conseillons vivement de sacrifier le temps nécessaire à la parfaite compréhension de ce principe.

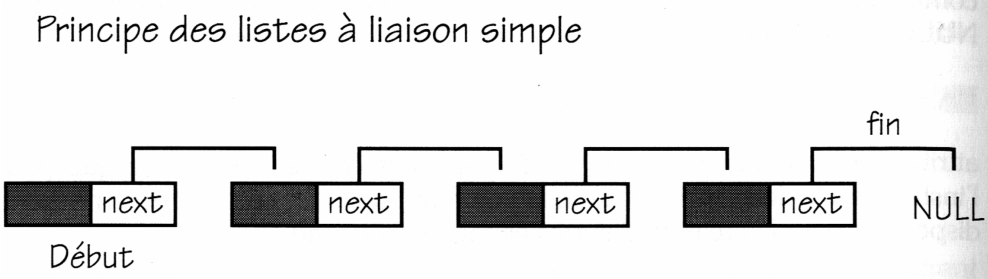
Lorsque l'exécution de la boucle est achevée, nous disposons donc d'une liste de sept éléments de même structure, compte tenu du premier non généré par la boucle. Cette liste possède les propriétés suivantes :

Le pointeur *start* est dirigé sur le premier élément de la liste.

Chaque élément de la liste contient un pointeur dirigé sur l'élément qui suit.

Le pointeur du dernier élément créé est dirigé sur NULL, et peut donc être utilisé pour reconnaître la fin de la liste.

Cela signifie que dans la liste vous ne pouvez sauter par dessus aucun élément : pour accéder à un élément de la liste, il faut passer par tous ceux qui le précèdent. Par conséquent, dès que vous désirez modifier des valeurs dans la troisième structure, il faut partir de la première et avancer progressivement jusqu'à la troisième.



Principe des listes à simple chaînage

Cette technique peut sembler longue, mais elle est excellente pour mémoriser et gérer des données dynamiques. Les traitements de textes constituent un parfait exemple de l'utilisation de listes liées, chaque ligne représentant un élément de la liste. Des programmes plus exigeants peuvent faire appel à des listes à double liaison : la structure contient alors un pointeur dirigé sur la ligne suivante, et un autre pointant sur la ligne précédente. Les données peuvent ainsi être parcourues dans les deux sens. Dans la section 14.5, nous verrons comment sont conçues les listes à double liaison.

14.3.4. Accès aux éléments d'une liste

Pour le renvoi des données mémorisées dans la liste a été programmée une boucle *do..while* dans les lignes 48 à 59. Son principe de fonctionnement est semblable à celui de la boucle qui avait généré les données. La boucle est précédée par l'initialisation d'un pointeur. Le pointeur *pointeur_courant* est dirigé sur l'origine de la liste, dont l'adresse est mémorisée dans *start*, et *fin* pointe sur l'élément suivant de la liste. Ainsi, tous les éléments de la liste sont parcourus jusqu'à ce que le pointeur *fin* soit dirigé sur NULL, c'est-à-dire l'extrémité de la liste. Le grand avantage de cette méthode réside dans le fait que le programme n'a pas besoin de connaître le nombre des éléments à imprimer. La sortie se poursuit jusqu'à ce qu'aucun élément ne puisse plus être détecté, c'est-à-dire jusqu'au moment où le pointeur *fin* soit dirigé sur NULL.

Il n'est pas difficile d'ajouter de nouveaux éléments à une liste. Il suffit à cet effet de réserver un nouveau bloc de mémoire, d'y entrer des données, et de faire en sorte que son pointeur soit dirigé sur l'élément suivant. Si le nouvel élément doit être inséré entre les troisième et quatrième enregistrements existants, le pointeur de l'élément à insérer doit être dirigé sur l'élément 4, et celui du troisième

élément sur le nouvel élément. Dans la section 14.4, vous trouverez un exemple de fonction qui traite de ce cas.

14.3.5. Libération des variables dynamiques

Au début de ce chapitre, nous vous avions laissé entendre que les blocs de mémoire créés dynamiquement peuvent à nouveau être libérés aux fins d'autres utilisations. La fonction qui assume cette tâche est *free()*. Elle requiert comme argument le pointeur dirigé sur une variable dynamique. Nous ne pouvons donc libérer que les blocs de mémoire sur lesquels est dirigé un pointeur. La technique de programmation appropriée nous est présentée dans les lignes 64 à 70. Dans cette partie du programme, la plage de mémoire occupée par les éléments de notre liste est à nouveau libérée. Ici aussi, nous considérons d'abord l'élément dont l'adresse est *start*, et parcourons la liste jusqu'à ce que nous rencontrions l'adresse NULL.

14.4. Listes à double chaînage

Dans les listes à liaison double, chaque élément possède deux pointeurs : l'un est dirigé sur l'élément suivant, et l'autre sur l'élément précédent. Le programme de démonstration LISTE.C vous permettra de découvrir des fonctions grâce auxquelles il est possible de se déplacer librement dans une liste, d'y ajouter des éléments (à l'extrémité, ou en plein milieu), de localiser et d'effacer des éléments.

Listing 70 : Le programme LISTE.C

```
[1 ] /* Chapitre 14.4 */
[2 ] /* Nom du programme : LISTE.C */
[3 ] /* Programme de démonstration - Fonctions de chaînage de listes */
[4 ]
[5 ]
[6 ] int append(void);              /* Si 1 alors réussi, si 0 erreur */
[7 ] int insert(int pos);           /* insérer à la position pos */
[8 ] int positionner(int pos);      /* positionner sur pos*/
[9 ] int delete(int pos);           /* Effacer un élément de la liste */
[10] void croissant(void);          /* Affichage de la liste dans le sens
                                       croissant */
[11] void decroissant(void);        /* Affichage de la liste dans le sens
                                       décroissant */
[12]
[13]
[14] #include <stdio.h>
```

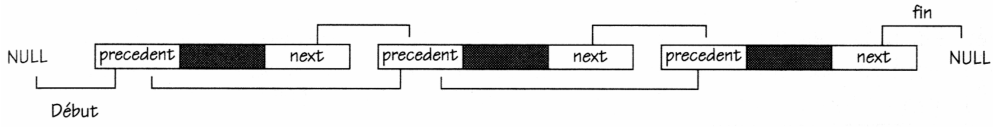
```
[15]
[16] struct  element
[17] {
[18]   struct element *precedent;     /* Pointeur sur élément précédent */
[19]   char string[25];
[20]   int numero;
[21]   struct element *next;          /* Pointeur sur élément suivant */
[22] } *fin, *act, *start;
[23]
[24] int numenreg,
[25] temp,
[26] l = sizeof(struct element);      /* Capacité mémoire requise */
[27]                                  /* par l'élément            */
[28] main()
[29] {
[30]   int i;                         /* Compteur pour la boucle */
[31]
[32]   start = (struct element *) malloc(l);    /* Réservation d'espace
                                                   mémoire*/
[33]   fin = (struct element *) malloc(l);
[34]
[35]   start->precedent = NULL;       /* Adresse de départ mise à NULL     */
[36]   start->next = fin;
[37]   fin->precedent = start;        /* et adresse de fin vaut aussi NULL */
[38]   fin->next = NULL;              /* pas encore d'élément              */
[39]   fin = start;
[40]
[41]   strcpy(start->string,"Le premier nom");
[42]   start->numero = 1;
[43]   numenreg = 1;
[44]
[45]   /* Voici 3 nouveaux enregistrements */
[46]   for (i = 1; i < 4; i++)
[47]       (append());
[48]
[49]   decroissant();                 /* Liste en arrière */
[50]
[51]   /* Insérer maintenant un nouvel élément à la position 2 */
[52]
[53]   insert(2);
[54]   croissant(); /* Nouvel affichage, mais cette fois en avant, */
[55]                /* et avec le nouvel élément                   */
[56]
[57]   /* Renvoi du deuxième élément */
[58]   if (positionner(2))
[59]       printf("\nLe nom recherché est \t%s et possède le numéro %d",
[60]       act->string,act->numero);
[61]   else
[62]       printf("\nNuméro d'enregistrement en dehors de la liste !!");
[63]
[64]   if (delete(2))
[65]     printf("\nDeuxième élément retiré de la liste ");
[66]
[67]   croissant();
[68] } /* et Fin du programme */
```

```
[69]
[70] int append(void)
[71] {
[72]   if ( ! (act = (struct element *) malloc(l)) )
[73]   return(0);              /* Plus de mémoire disponible */
[74]
[75]   numenreg++;
[76]   fin->next = act;
[77]   act->precedent = fin;
[78]   printf("\nVeuillez entrer un nom : ");
[79]   scanf("%s", act->string);
[80]   act->numero = numenreg;
[81]   act->next = NULL;
[82]   fin = act;
[83]   return(1);              /* Exécution de fonction réussie */
[84] }
[85]
[86] int insert(int pos)       /* Insérer un élément dans la liste à la
                                  position pos */
[87] {
[88]
[89]   struct element *temp;
[90]
[91]   temp = (struct element *) malloc(l);
[92]
[93]   if (!(positionner(pos)))
[94]       return(0);          /* Erreur, d'où retour */
[95]
[96]   temp->precedent = act->precedent;
[97]   act->precedent->next = temp;
[98]   temp->next = act;
[99]   act->precedent = temp;
[100]   strcpy(temp->string,"Nouvel élément"); /* Affectation de valeur */
[101]   temp->numero = 99;
[102]
[103]   numenreg++;          /* Incrémentation du numéro d'enregistrement */
[104]   return(1);           /* puis retour */
[105] }
[106]
[107] int delete(int pos)    /* Insérer un élément dans la liste à la
                                  position pos */
[108] {
[109]   if (!positionner(pos))
[110]       return(0);       /* Numéro de champ en dehors de la liste, */
[111]                        /* d'où retour                            */
[112]
[113]   act->precedent->next = act->next;    /* Echange des adresses */
[114]   act->next->precedent = act->precedent;
[115]
[116]   free(act);          /* puis libération de l'espace mémoire */
[117]   numenreg--;         /* un élément de moins                 */
[118]   return(1);          /* et retour                           */
[119] }
[120]
[121] int positionner(int pos) /* Positionnement sur l'élément */
```

```
[122] {
[123]   int i;
[124]   act = start;
[125]   for (i = 1; i < pos;i++)
[126]   {
[127]       if ( (act = act->next) == NULL)
[128]         return(0);     /* Retour : Code d'erreur 0 = faux */
[129]   } /* Positionnement à la position pos */
[130]   return(1);      /* et retour - Exécution de fonction réussie */
[131] }
[132]
[133] void decroissant(void)     /* Liste en sens inverse */
[134] {
[135]   temp = numenreg;
[136]   act = fin;               /* Positionner sur fin de liste */
[137]   printf("Numéro de liste\t\Numéro d'enregistrement\tNom");
                                  /* Titre */
[138]
[139]   while (act != NULL)      /* tant que non début de la liste */
[140]   {
[141]     printf("\n%d\t\t\t%d\t%s",temp,act->numero, act->string);
[142]                            /* renvoyer le contenu de l'élément */
[143]     temp--;                /* Décrémenter le compteur */
[144]     act = act->precedent; /* Adresse de la liste précédente  */
[145]   }                        /* Fin de la boucle d'affichage */
[146] }                          /* Fin de la fonction */
[147]
[148] void croissant(void)
[149] {
[150]   printf("\n");
[151]   temp = 1;             /* Le numéro de la liste est 1 */
[152]   act = start;          /* commençons par start, le premier élément */
[153]   while( act != NULL)   /* tant que non fin de la liste */
[154]   {
[155]    printf("\n%d\t\t\t%d\t%s",temp,act->numero, act->string);
[156]    temp++;
[157]    act = act->next;     /* élément suivant */
[158]   }                     /* Fin boucle d'affichage */
[159] } /* Fin de la fonction */
```

Dans les lignes 6 à 10, vous voyez les prototypes des fonctions utilisées dans notre programme de démonstration. Vient ensuite la déclaration de notre structure *element*, composée d'une chaîne, d'une variable entière, et de deux pointeurs dirigés sur la structure. Le premier pointeur, dont le nom est *precedent*, nous servira à pointer sur l'élément situé juste devant l'élément courant. L'adresse de l'élément précédent y sera mémorisée. Le pointeur next est dirigé sur l'élément qui suit, dans la liste, dont il contient donc l'adresse.

Principe des listes à liaison double



Principe d'une liste à double chaînage

Dans la ligne 22 nous déclarons trois pointeurs dirigés sur la structure *element* : *fin*, *act* et *start*. Notez qu'une fois de plus il ne s'agit pas là de variables ! Nous disposons simplement de trois pointeurs susceptibles d'être dirigés sur des données de type *element*. Les tâches prises en charge par ces pointeurs dans le programme sont les suivantes :

Le pointeur *start* est dirigé sur le premier élément de la liste.

② Le pointeur *act* est dirigé sur l'élément courant de la liste, dont le traitement est en cours.

③ Le pointeur *fin* est dirigé sur le dernier élément de la liste.

Dans la ligne 26, la capacité de mémoire nécessaire à la structure *struct element* est affectée à la variable entière *l*. Nous verrons par la suite que cette capacité sera systématiquement nécessaire pour le remplissage de nouveaux blocs de mémoire sur le tas. Dans l'immédiat, nous allons analyser le fonctionnement du programme principal *main()*, puis nous consacrer intensivement aux fonctions qui y sont programmées.

Dès que vous lancez notre programme de démonstration, il vous invite à entrer trois noms, pour trois éléments de la liste. A l'aide de la fonction *append()* appelée trois fois dans la boucle *for* des lignes 46 et 47, ces éléments sont ajoutés à la fin de la liste existante. Suite à cela, vous voyez apparaître à l'écran les lignes suivantes :

Affichage retourné par le programme LISTE.C

```
Veuillez entrer un nom : Nom1
Veuillez entrer un nom : Nom2
Veuillez entrer un nom : Nom3
Numéro dans la liste     Numéro d'enregistrement     Nom
4                        4                           Nom3
```

```
3                    3                    Nom2
2                    2                    Nom1
1                    1                    Monnet
```

```
1                    1                    Monnet
2                    99                   Nouvel élément
3                    2                    Nom1
4                    3                    Nom2
5                    4                    Nom3
```

```
Le nom recherché est    Nouvel élément et possède le numéro 99
Deuxième élément retiré de la liste
```

```
1                    1                    Monnet
2                    2                    Nom1
3                    3                    Nom2
4                    4                    Nom3
```

Dès que les trois éléments ont été chargés, les éléments de la liste sont renvoyés, dans l'ordre inverse de leur saisie, à l'aide de la fonction *decroissant()* de la ligne 49. Dans un second bloc de renvoi, l'affichage est réalisé dans l'ordre chronologique réel, en commençant par le premier élément de la liste, et en finissant par le dernier. La fonction utilisée à cet effet se trouve en ligne 54, et son nom est *croissant()*. Entre temps, un nouvel élément avait été inséré en deuxième position dans notre liste à l'aide de la fonction *insert(2)*. Il en résulte une discordance entre le numéro d'ordre dans la liste et le numéro de l'enregistrement. L'élément dont le numéro d'enregistrement est 99 a en effet été inséré ultérieurement dans la liste, à la position 2.

Dans la partie suivante du programme, l'on se positionne sur l'élément dont le numéro d'ordre dans la liste est 2, par l'intermédiaire de la fonction *positionner(2)* qui figure en ligne 58. Dans la ligne 64, nous effaçons l'élément qui occupe le position 2, à l'aide de *delete(2)*, puis relançons l'affichage de la liste au moyen de *croissant()*. Comme vous pouvez le constater, nous avons maintenant sous les yeux la liste des noms que vous aviez vous-même entrés, dans laquelle le numéro d'ordre dans la liste et le numéro d'enregistrement sont à nouveau les mêmes.

Nous venons donc d'ajouter des éléments en fin de liste, d'en insérer dans la liste, d'en rechercher et d'en effacer. Ces opérations ont été exclusivement réalisées grâce à la double liaison des éléments de la liste. Analysons maintenant séparément comment ont été programmés ces différents processus.

14.4.1. La fonction append() - Ajouter des éléments en fin de liste

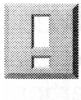
La tâche de la fonction *append()* programmée dans les lignes 70 à 84 est d'ajouter de nouveaux éléments à la fin d'une liste. Dans ce but, la fonction vérifie d'abord si la mémoire encore disponible sur le tas est suffisante. La fonction *malloc()* renvoie 0, et est donc fausse dans la logique du C, lorsqu'il n'est pas possible de réserver suffisamment de mémoire. Si tel est le cas, la fonction est interrompue avant terme en ligne 73, et la valeur 0 est retournée à la fonction appelante. Si ce n'est pas le cas, nous augmentons le numéro de l'enregistrement en ligne 75, et attribuons au pointeur *fin->next* l'adresse du nouveau bloc de mémoire, c'est-à-dire celle de *act*. Par conséquent, *fin->next* ne pointe plus sur NULL, mais sur le nouvel élément.

Dans l'étape suivante, en ligne 77, l'adresse de *fin* est affectée à *act->precedent*. Il s'en suit que le dernier élément, c'est-à-dire *act*, contient dès lors l'adresse de l'avant dernier élément, c'est-à-dire *fin*.

A présent, dans les lignes 79 à 80, des valeurs sont attribuées aux variables *act->string* et *act->numero*.

Les lignes 81 et 82 qui viennent sont très importantes et vont être déterminantes. Dans un premier temps, nous attribuons la valeur NULL au pointeur *act->next*. Ainsi, *act* est désigné comme étant le dernier élément de la liste. Il n'est plus suivi que par NULL, qui signale qu'il n'y a plus d'autre élément. Ensuite, nous attribuons au pointeur *fin*, qui se rapporte au dernier élément de la liste, l'adresse mémorisée dans *act*. Dès lors, *act* et *fin* pointent tous deux sur le dernier élément de la liste. Le pointeur *act* redevient disponible pour une nouvelle adresse.

Suite à l'ajout réussi d'un élément à la liste, la fonction *append()* s'achève en retournant la valeur 1. Nous signalons ainsi à la fonction appelante que l'ajout d'un nouvel élément s'est déroulé avec succès.



Par conséquent, les étapes nécessaires pour ajouter un élément à la fin d'une liste sont les suivantes :

① Réserver sur le tas une plage de mémoire appropriée dont l'adresse de l'origine doit être attribuée au pointeur *act*.

② Transmettre au pointeur *fin->next* l'adresse du nouvel élément.

③ Transmettre au pointeur *act->precedent* l'adresse de fin.

④ Attribuer des valeurs.

⑤ Attribuer au pointeur *act->next* la valeur NULL.

⑥ Attribuer au pointeur *fin* le contenu de *act*, c'est-à-dire l'adresse du nouvel élément qui, à cet instant, est le dernier de la liste.

Nous pouvons ainsi créer de nouveaux éléments aussi longtemps que la mémoire disponible sur le tas n'est pas épuisée. Chaque élément est relié aux autres par l'intermédiaire des pointeurs adéquats. Dans la section qui suit, nous allons voir comment la fonction *insert()* permet d'intégrer des éléments à un emplacement quelconque dans la liste.

14.4.2. La fonction insert() - Insérer des éléments dans une liste

Pour insérer un élément à l'intérieur d'une liste, deux opérations sont nécessaires. Il faut d'abord se positionner sur l'élément à la suite duquel le nouvel élément doit être inséré. Ensuite, il faut redéfinir les pointeurs du nouvel élément, ceux de l'élément qui précède, et ceux de l'élément qui le suivra dans la liste. La technique de programmation de ces opérations nous est présentée par la fonction *insert()*, dans les lignes 86 à 105.

Dans la ligne 89, nous déclarons le pointeur *temp* dirigé sur le type de données *element*. Dans la suite du déroulement de la fonction, nous utiliserons ce pointeur pour insérer les nouveaux éléments dans la liste.

Mais dans l'immédiat, il nous faut d'abord accéder à la position d'insertion. Nous utilisons à cet effet la fonction *positionner()*, qui est programmée dans les lignes 121 à 131. Dans la ligne 124, nous attribuons au pointeur *act* l'adresse issue de *start*. Nous nous positionnons donc sur l'origine de la liste. Dans la boucle *for* qui suit, et qui sera parcourue jusqu'à ce que le nombre de cycles soit égal à la valeur de la variable *pos*, nous nous positionnons sur le point d'insertion. La ligne

```
if ( (act = act->next) == NULL)
```

fait en sorte que l'adresse de l'élément suivant, mémorisée dans *act->next*, soit d'abord affectée au pointeur *act*. Le programme vérifie ensuite si cette adresse est égale à NULL. Si tel était le cas, c'est qu'il n'existerait aucun élément après *act* dans la liste. Aucun élément ne pourrait donc être inséré à cet emplacement, ni à un endroit plus éloigné. C'est la raison pour laquelle la fonction serait interrompue en ligne 128, en retournant la valeur 0 à la fonction appelante. Par contre, s'il existe un élément à la suite de *act*, et si après incrémentation la valeur de *i* est encore inférieure à *pos*, la boucle sera encore parcourue une fois. En fin d'exécution de boucle, le pointeur *act* contient alors l'adresse de l'élément de la liste à l'emplacement duquel le nouvel élément doit être inséré.

Lorsque le positionnement a réussi, la fonction *positionner()* renvoie la valeur 1 à la fonction appelante, c'est-à-dire, dans notre cas, à *insert()*.

Dans les lignes qui suivent l'activation de *positionner()*, et qui, comme nous le savons, ne sont parcourues que si le positionnement a réussi, le programme redéfinit maintenant les adresses de renvoi des éléments concernés de la liste. Avant d'approfondir cette analyse, établissons un rapide bilan de la situation actuelle :

① Le pointeur *act* est dirigé sur l'élément à la position duquel le nouvel élément doit être inséré.

② Le pointeur *temp* est dirigé sur une zone de mémoire libre sur le tas.

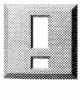
La ligne 96

```
temp->precedent = act->precedent;
```

attribue maintenant au pointeur *precedent* de *temp* l'adresse provenant de *act->precedent*. Dès lors, *temp* connaît lui aussi l'adresse de l'élément situé juste devant le point d'insertion. Dans la ligne 97

```
act->precedent->next = temp;
```

l'adresse de *temp* est transmise au pointeur *next* de l'élément situé devant le point d'insertion, et dont l'adresse est *act->precedent*.



Pour bien comprendre ce processus, il est important de connaître parfaitement la signification de la désignation de variable

```
act->precedent->next
```

Nous avons à faire ici à un double renvoi de pointeurs. Le nom de la variable fait référence au contenu du pointeur *next*, sur lequel est dirigé le pointeur *precedent*. Et sur ce dernier est dirigé le pointeur *act*. Par conséquent, *act->precedent->next* désigne le pointeur de l'élément précédent, dans lequel est mémorisée l'adresse du suivant.

A présent, l'élément placé juste devant le point d'insertion connaît l'adresse de *temp* par l'intermédiaire du pointeur *next*. Et réciproquement, *temp* connaît, par l'intermédiaire du pointeur *precedent*, l'adresse de cet élément.

Dans les lignes 97 et 98 qui suivent, le programme établit la liaison avec l'élément suivant de la liste. Dans ce but, l'adresse de *act* est attribuée au pointeur *temp->next* comme adresse de l'élément suivant, puis l'adresse mémorisée dans *temp* est attribuée au pointeur *act->precedent* en guise d'adresse de l'élément précédent.

A présent, *temp* est inséré à l'emplacement où se trouvait l'élément possédant l'adresse mémorisée dans *act*. Dans les lignes qui suivent, des valeurs sont attribuées aux variables de *temp*, le compteur des numéros d'enregistrements est incrémenté, et la fonction s'achève en retournant la valeur 1.

Le domaine de validité du principe que vous venez de découvrir avec la fonction *insert()* est général : quelle que soit la structure avec laquelle vous travaillez, la logique de l'insertion dans une liste à double liaison est la même. Etant donné que la bonne compréhension de cette technique est indispensable pour tout ce qui touche au traitement de listes, il est bon de récapituler les différentes étapes du processus :

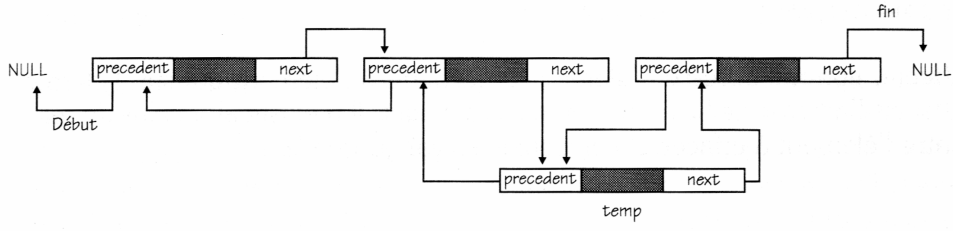
① Positionnement sur le point d'insertion

② Le nouvel élément reprend dans l'élément situé au point d'insertion l'adresse de celui qui précède.

③ L'élément précédent prend comme adresse de l'élément suivant celle de l'élément à insérer.

④ Le nouvel élément prend comme adresse de l'élément suivant celle de l'élément situé au point d'insertion.

⑤ L'élément situé au point d'insertion prend comme adresse de l'élément précédent celle du nouvel élément à insérer.



Insertion d'éléments dans une liste

Dans la section qui suit, dans laquelle nous apprendrons à supprimer des éléments d'une liste, vous verrez que l'insertion et l'effacement d'éléments présentent certaines similitudes.

14.4.3. La fonction delete() - Supprimer des éléments d'une liste

Deux principes fondamentaux régissent aussi la suppression d'éléments d'une liste : d'abord se positionner, ensuite modifier les adresses de renvoi. Un effacement dans une liste liée consiste à lever les liaisons établies par l'intermédiaire des adresses mémorisées dans l'élément à effacer, puis à libérer l'espace mémoire occupé par cet élément.

Cette technique est utilisée dans les lignes 107 à 119, la ligne 113 étant la plus importante :

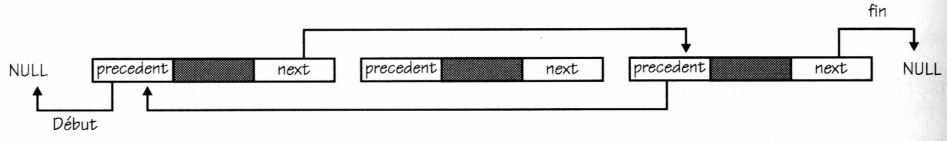
```
act->precedent->next = act->next;
```

Cette ligne signifie que l'adresse contenue dans l'élément, qui précède l'élément à effacer, et qui se réfère à cet élément, sera modifiée. Dans *next*, la nouvelle adresse de l'élément suivant qui sera affectée à l'élément précédant celui à effacer, est l'adresse de l'élément suivant incluse dans l'élément à effacer. L'élément situé devant *act* ne connaît maintenant plus l'adresse de *act*, mais celle de l'élément qui suit *act*.

Dans la ligne 114, l'instruction

```
act->next->precedent = act->precedent;
```

affecte, comme adresse d'élément précédent, à l'élément qui suit l'élément à effacer, l'adresse de l'élément précédent de *act*. Il n'existe alors plus de relation entre l'élément à effacer et celui qui le suivait jusque-là.



Effacement d'éléments dans une liste

A la suite de ce transfert d'adresses, il n'existe plus aucune liaison entre la liste et l'élément à effacer. C'est la raison pour laquelle, dans la ligne 116, nous libérons la plage de mémoire occupée par cet élément et décrémentons le compteur d'enregistrements. Une fois de plus, nous signalons le succès de l'opération (d'effacement) par le renvoi de la valeur 1.

14.4.4. Les fonctions croissant() et decroissant() - Défilement d'une liste

Les fonctions *croissant()* et *decroissant()* que nous allons décrire montrent comment, dans une liste à double liaison, il est possible de se déplacer. La logique des deux fonctions étant la même, nous avons regroupé leur description dans une même rubrique. La seule différence réside dans le fait que, pour la fonction *croissant()*, nous commençons avec le premier élément de la liste, alors que pour *decroissant()* nous commençons au dernier. C'est la raison pour laquelle dans la ligne 152 le pointeur *act* est initialisé avec l'adresse mémorisée dans *start*, et dans la ligne 136 avec celle contenue dans *fin*.

Pour les deux fonctions, la condition d'exécution de boucle est la même. Dans la fonction *croissant()*,

```
while ( act != NULL)
```

est vrai tant que *act->next* ne pointe pas sur NULL. De même, pour la fonction *decroissant()*, la condition d'exécution de boucle est vraie tant que *act->precedent* ne pointe pas sur NULL.

Lorsque l'on parcourt la liste vers l'avant, l'adresse de l'élément suivant mémorisée dans *act->next* est attribuée au pointeur courant après chaque cycle de la boucle. Lorsque l'on parcourt la liste en arrière, c'est l'adresse de l'élément précédent qui est transmise au pointeur courant. La fonction *croissant()* assure par conséquent un déplacement vers la fin de la liste, et la fonction *decroissant()* vers le début de cette liste.

Dans les deux fonctions, la variable *temp* a pour tâche de renvoyer pour chaque élément son numéro d'ordre dans la liste. Elle ne joue aucun rôle pour les mouvements dans la liste.

14.5. Récapitulation

Les variables dynamiques sont des variables qui ne sont créées que pendant l'exécution d'un programme, et qui peuvent être libérées après utilisation. Les variables dynamiques sont mémorisées dans une plage de mémoire appelée tas.

La fonction *malloc()* est celle qui permet de réserver une plage de mémoire sur le tas pour une variable dynamique. Le résultat renvoyé par *malloc()* est un pointeur dirigé sur la plage de mémoire réservée. La fonction *free()* permet de libérer cette plage de mémoire, afin de la rendre disponible pour d'autres variables. En guise d'argument, cette fonction requiert le pointeur dirigé sur la zone de mémoire appropriée. La gestion dynamique de la mémoire est une technique qui permet de gérer un très grand nombre de données dans un même programme.

Les listes à liaison simple sont des listes de variables structurées générées dynamiquement. Chaque élément est en relation avec son suivant par l'intermédiaire d'un pointeur qui contient l'adresse de l'élément suivant. Le pointeur intégré au dernier élément d'une liste est dirigé sur NULL, signalant ainsi la fin de la liste. Pour accéder à un élément de la liste, il est nécessaire de passer par tous ceux qui le précèdent. Pour pouvoir travailler avec des listes liées, il faut que soit connue l'adresse du premier élément de la liste, et que le dernier élément de cette liste pointe sur NULL. Les éléments extrêmes jouent donc un rôle particulier.

Les listes à liaison double sont caractérisées par l'existence d'une relation entre l'élément suivant et l'élément précédent, au moyen de pointeurs dans lesquels sont mémorisées les adresses correspondantes. De telles listes peuvent être parcourues dans les deux sens. Dans le premier élément des listes à liaison double, le pointeur dirigé sur l'élément précédent pointe sur NULL. Dans le dernier élément de ces listes, c'est le pointeur dirigé sur l'élément suivant qui pointe sur NULL. Un défilement dans une telle liste se poursuit, quel qu'en soit le sens, jusqu'à ce qu'une adresse de renvoi soit égale à NULL. Lorsque l'on ajoute des éléments à une liste, ou que l'on en retire, il faut modifier les adresses mémorisées pour l'élément précédent et sur l'élément suivant.

L'avantage des listes liées réside dans le fait qu'il n'est pas nécessaire de connaître le nombre d'éléments qu'elles contiennent.

Exercices se rapportant au chapitre 14

Exercice 1 :
Ecrivez un programme qui, à l'aide de la fonction

```
int input(void)
```

que vous programmerez, charge un nombre quelconque de valeurs entières dans une liste à générer dynamiquement, et qui écrive ces valeurs dans un fichier à l'aide d'une fonction

```
int output(FILE *pointeurfichier)
```

Vous programmerez ces deux fonctions de manière à ce qu'elles retournent la valeur 1.

La saisie de la valeur 0 indiquera la fin de la liste.

Demandez ensuite que le contenu du fichier ainsi écrit soit visualisé à l'écran.

Exercice 2 :
Modifiez le programme STRUCT2.C de manière à ce que le tableau enfants[12] soit généré dynamiquement.

15 Manipulation de caractères et de bits

Dans ce chapitre vous apprendrez comment classifier et modifier des caractères, à l'aide des fonctions standard disponibles en C. Vous découvrirez la multiplication et la division rapides à l'aide des opérateurs Shift gauche et Shift droite, ainsi que l'utilisation d'opérateurs logiques agissant sur des bits. En conclusion de ce chapitre vous trouverez un exemple de programme qui montre l'accès direct à des structures de bits.

15.1. Classification de caractères

Le programme MAJMIN.C est l'exemple de manipulation de caractères que nous avons choisi. Dans ce cas particulier, il convertit les minuscules en majuscules et vice-versa. Le programme extrait toutes les lignes d'un fichier et, en une seule fonction, remplace les majuscules par des minuscules et les minuscules par des majuscules.

Listing 71 : Le programme MAJMIN.C

```
[1 ] /* Chapitre 15.1 */
[2 ] /* Nom du programme: MAJMIN.C */
[3 ] /* Exemple de manipulation de caractères en C */
[4 ] /* Fonctions:    isupper() */
[5 ] /*               islower() */
[6 ] /*               tolower() */
[7 ] /*               toupper() */
[8 ]
[9 ] #include "STDIO.H"
```

```
[10] #include "ctype.h"      /* constantes et macros pour manipulations de
                                  caractères */
[11]                        /* Définitions pour isupper(), islower() */
[12]
[13] main()
[14] {
[15]   FILE *fp;
[16]   char ligne[80], nomfichier[24];
[17]   char *c;
[18]   printf("Entrez le nom du fichier -> ");
[19]   scanf("%s",nomfichier);
[20]   fp = fopen(nomfichier,"r");
[21]
[22]   do
[23]   {
[24]     c = fgets(ligne,80,fp);     /* charger une ligne de texte */
[25]     if (c != NULL)              /* Lecture réussie, alors     */
[26]       echange_majus_minus(ligne);
[27]                                 /* Activation de la fonction  */
[28]   } while (c != NULL);          /* jusqu'à la fin du fichier  */
[29]
[30]   fclose(fp);
[31] }
[32]
[33] echange_majus_minus(ligne) /* Remplace les majuscules                */
[34]                            /* par des minuscules et réciproquement, */
[35]                            /* en ignorant les autres caractères     */
[36] char ligne[];              /* Style de programmation classique      */
[37] { /* Début de fonction */
[38]     int index;
[39]
[40]   for (index = 0;ligne[index] != 0;index++)
[41]   {
[42]     if (isupper(ligne[index]))                  /* 1. majuscule */
[43]       ligne[index] = tolower(ligne[index]);     /* conversion   */
[44]     else
[45]     {
[46]       if (islower(ligne[index]))                /* 2. minuscule */
[47]         ligne[index] = toupper(ligne[index]);   /* conversion   */
[48]     }
[49]   }
[50]   printf("%s",ligne);
[51] }
```

Au début du programme, dans la ligne 10, il y a intégration du fichier d'en-tête CTYPE.H. Celui-ci contient deux macros qui nous permettront de détecter les majuscules *(isupper())*, et les minuscules *(islower())*.

La logique du programme n'a rien de nouveau. C'est pourquoi nous vous engageons à porter toute votre attention sur la fonction

```
echange_majus_minus(ligne);
```

Dans les lignes 40 à 49 est parcourue une boucle qui traite tous les caractères d'une ligne transmise à la fonction. A cette occasion, dans la ligne 42, le programme vérifie d'abord si le caractère courant *ligne[index]* est une majuscule, par l'intermédiaire de la ligne :

```
if (isupper(ligne[index]))
```

Si c'est le cas, ce caractère sera transformé en une minuscule au moyen de la fonction *tolower()*. Dans le cas contraire, le programme vérifie, dans la ramification else de l'instruction *if..else*, si le caractère courant est une minuscule. Dans l'affirmative, il sera converti en majuscule à l'aide de la fonction *toupper()*.

Il s'en suit que tous les caractères du fichier ouvert qui sont des lettres seront convertis. Tous les autres caractères restent inchangés. Les nouvelles fonctions assurent donc les tâches suivantes :

```
isupper()  vrai, si le caractère est une majuscule
islower()  vrai, si le caractère est une minuscule
toupper()  transforme en majuscule
tolower()  transforme en minuscule
```

Vous trouverez d'autres routines de classification dans le guide de référence de votre compilateur. Pour ce qui concerne les compilateurs dont il est question dans ce livre, il s'agit de : *isalpha()*, *isdigit()*, *islower()*, *isspace()*, *isupper()*, *toupper()* et *tolower()*.

Le programme CARACTC.C qui suit nous servira à montrer comment une répartition de caractères peut être réalisée en C. Ce programme "lit" un fichier, et compte les caractères alphanumériques, numériques et particuliers. Sont considérés comme caractères particuliers les espaces (blancs), les tabulations et les sauts de lignes.

Listing 72 : Le programme CARACTC.C

```
[1 ] /* Chapitre 15.1 */
[2 ]
[3 ] /* Nom du programme: CARACTC.C */
[4 ] /* Autre exemple de manipulations de caractères */
[5 ]
[6 ] #include "stdio.h"
[7 ] #include "ctype.h"
[8 ]
[9 ] main()
[10] {
[11]   FILE *fp;
[12]   char ligne[80], nomfichier[24];
[13]   char *c;
```

```
[14]
[15]   printf("Entrez un nom de fichier -> ");
[16]   scanf("%s",nomfichier);
[17]   fp = fopen(nomfichier,"r");
[18]
[19]   do
[20]   {
[21]     c = fgets(ligne,80,fp);     /* Lis une ligne */
[22]     if (c != NULL)
[23]       compte_les_caracteres(ligne);
[24]   } while (c != NULL);
[25]
[26]   fclose(fp);
[27] }
[28]
[29] compte_les_caracteres(ligne)
[30] char ligne[];
[31] {
[32]   int semi_graphiques, caracteres, chiffres;
[33]   int index;
[34]   semi_graphiques = caracteres = chiffres = 0;
[35]
[36]   for (index = 0;ligne[index] != 0;index++)
[37]   {
[38]       if (isalpha(ligne[index])) /* 1. ligne[] alphabétique */
[39]        caracteres++;
[40]     if (isdigit(ligne[index]))   /* 2. ligne[] chiffre      */
[41]       chiffres++;
[42]     if (isspace(ligne[index]))   /* 3. ligne[] Blanc, Tabulateur,
                                          Saut de ligne */
[43]       semi_graphiques++;
[44]   }  /* Fin de boucle de comptage */
[45]
[46]   printf("%3d caractères semi-graphiques \t
            %3d caractères alphabétiques \t%3d chiffres dans \n%s\n",
[47]          semi_graphiques,caracteres,chiffres,ligne);
[48] }
```

Ce programme contient, lui aussi, une fonction écrite par le programmeur. Il s'agit de

```
compte_les_caracteres(ligne);
```

qui utilise trois macros à l'aide desquelles il est possible de classifier tous les caractères d'un texte quelconque.

Ces macros, qui proviennent de CTYPE.H assurent les tâches suivantes :

```
isalpha() : vrai, si le caractère est une lettre
isdigit() : vrai, si le caractère est un chiffre
isspace() : vrai, si le caractère est un espace,
            une tabulation ou un symbole de saut de ligne.
```

Dans les programmes de démonstration sur lesquels nous avons travaillé jusqu'ici, nous utilisions systématiquement \n pour un changement de ligne. Mais le C connaît encore d'autres codes de contrôle, appelés séquences d'échappement (ou séquences escape). En voici la liste :

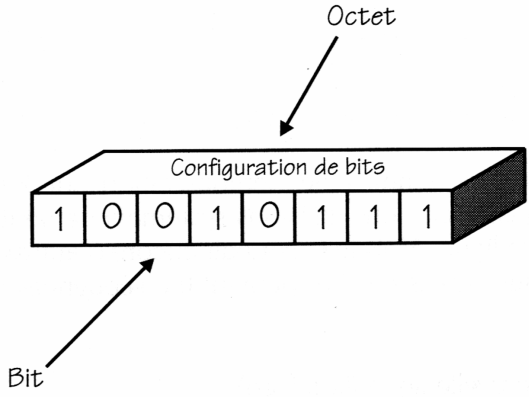
\n	Nouvelle ligne (line feed)
\t	Tabulation
\f	Saut de page (form feed)
\b	Backspace (retour arrière)
\r	Retour chariot (carriage return)
\"	Guillemet
\0	NULL

Comme vous le savez, les séquences d'échappement peuvent figurer en nombre illimité à des emplacements quelconques dans un texte.

Les programmes MAJMIN.C et CARACTC.C sont destinés à vous montrer un certain nombre de possibilités de manipulations de caractères en C. N'hésitez pas à expérimenter les fonctions et macros que nous y présentons. Vous verrez que les possibilités de manipulation de caractères sont multiples.

15.2. Opérateurs logiques pour la manipulation de bits

La plus petite unité adressable dans notre mémoire de travail est l'octet (byte). Chaque octet est composé de 8 bits, le bit étant la plus petite unité d'information pour un ordinateur. Chaque bit peut prendre deux valeurs que l'on représente généralement par 0 et 1. Lorsque la valeur d'un bit est 1, l'on dit que ce bit est placé. La combinaison des bits placés et non placés d'un octet est appelée configuration de bits.



Octet, bit et configuration de bits

Le langage C connaît 6 opérateurs agissant sur des bits. Dans cette section, vous ferez la connaissance des quatre opérateurs logiques s'appliquant aux bits, grâce au programme BITOPS.C. Ceux-ci permettent d'accéder directement à des configurations de bits, et à les modifier.

Opérateurs logiques s'appliquant aux bits, en C :

Symbole	Signification	Effet produit
&	ET logique	lorsque chaque bit vaut 1, le bit résultant est 1
|	OU logique	lorsque l'un des bits vaut 1, le bit résultant est 1
^	OU exclusif	lorsqu'un et un seul des bits vaut 1, le bit résultant est 1
~	NON logique (ou INVERSE)	lorsque le bit vaut 1, le bit résultant est 0, et lorsque le bit vaut 0, le bit résultant est 1

Listing 73 : Le programme BITOPS.C

```
[1 ] /* Chapitre 15.2 */
[2 ] /* Nom du programme: BITOPS.C */
[3 ] /* Exemple de manipulation de bits en C */
[4 ]
[5 ] main()
[6 ] {
[7 ]   char masque;
[8 ]   char nombre[6];
[9 ]   char and,or,xor,inv,index;
[10]
[11]   nombre[0] = 0X00;
[12]   nombre[1] = 0X11;
[13]   nombre[2] = 0X22;
[14]   nombre[3] = 0X44;
```

```
[15]   nombre[4] = 0X88;
[16]   nombre[5] = 0XFF;
[17]
[18]   printf(" nombre  masque   and    or   xor   inv\n");
[19]   masque = 0X0F;
[20]   for (index = 0;index <= 5;index++)
[21]    {
[22]       and = masque & nombre[index];
[23]       or = masque | nombre[index];
[24]       xor = masque ^ nombre[index];
[25]       inv = ~nombre[index];
[26]       printf("%5x %7x %6x %6x %5x %6x\n",
[27]              nombre[index], masque, and, or, xor,inv);
[28]     }
[29]
[30]   printf("\n");
[31]   masque = 0X22;
[32]   for (index = 0;index <= 5;index++)
[33]   {
[34]       and = masque & nombre[index];
[35]       or = masque | nombre[index];
[36]       xor = masque ^ nombre[index];
[37]       inv = ~nombre[index];
[38]       printf("%5x %7x %6x %6x %5x %6x\n",nombre[index],
[39]              masque,and,or,xor,inv);
[40]   }
[41] }
```

Ce programme fait appel aux opérateurs décrits précédemment. Il utilise des valeurs de variables au format hexadécimal qui, dans les lignes 11 à 16, sont attribuées aux éléments du tableau *nombre[]*. Dans les lignes 20 à 28 et 32 à 40, les valeurs des éléments de *index* sont mises en relation avec la valeur contenue dans la variable *masque* par l'intermédiaire des opérateurs &, | et ^. Les résultats ainsi produits sont alors respectivement attribués aux variables *and*, *or* et *xor*. Dans les lignes 25 et 37, le contenu de la variable *nombre[index]* est modifié directement à l'aide de l'opérateur unaire ~, et le résultat est transmis à la variable *inv*. La modification de la configuration de bits de la variable qui en résulte est ensuite renvoyée à l'écran.

Affichage produit par le programme BITOPS.C :

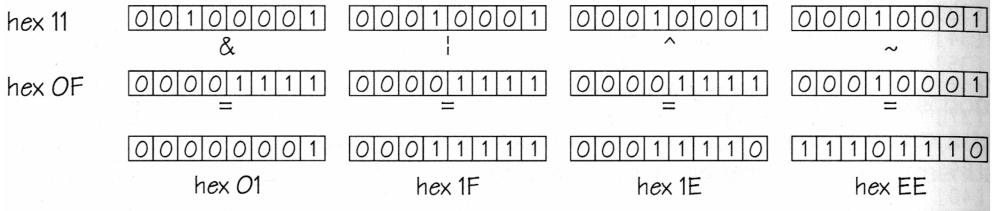
```
Valeur initiale            Relations
 nombre            masque  and    or   xor   inv
    0                 f     0     f     f  ffff
   11                 f     1    1f    1e  ffee
   22                 f     2    2f    2d  ffdd
   44                 f     4    4f    4b  ffbb
 ff88                 f     8  ff8f  ff87    77
 ffff                 f     f  ffff  fff0     0
    0                22     0    22    22  ffff
   11                22     0    33    33  ffee
```

```
   22          22    22    22     0  ffdd
   44          22     0    66    66  ffbb
 ff88          22     0  ffaa  ffaa    77
 ffff          22    22  ffff  ffdd     0
```

La représentation schématique qui suit montre les effets produits par affectations suivantes

```
and     = masque & nombre[1];
or      = masque | nombre[1];
xor     = masque ^ nombre[1];
inv     = ~nombre[index];
```

lorsque masque = hex F et nombre[1] = hex 11.



Opérateurs logiques s'appliquant à des bits

15.3. L'opérateur Shift

Dans cette section, vous découvrirez les deux opérateurs Shift gauche et droite. Ces opérateurs sont essentiellement utilisés lorsque des programmes doivent effectuer des calculs très rapidement. Le programme qui suit effectue la multiplication et la division de deux valeurs à l'aide des opérateurs Shift gauche et Shift droite.

Listing 74 : Le programme SHIFT.C

```
[1 ]    /* Chapitre 15.3 */
[2 ]    /* Nom du programme: SHIFT.C */
[3 ]    /* Exemple d'opérations avec Shift */
[4 ]
[5 ]    main()
[6 ]    {
[7 ]    unsigned int i,j,reduction, extension, index, compteur;
[8 ]
```

```
[9 ]    printf(" \t\t   Shift gauche\t\t\tShift droite      \n\n");
[10]   printf("        \t dec\t\t   hex\t        dec\t\t     hex\n");
[11]
[12]   reduction = i =1;
[13]   extension = j = 0x4000;
[14]   for(index = 0;index < 15;index++)
[15]   {
[16]       printf("\t%8u \t%8x \t%8u
                \t%8x\n",reduction,reduction,extension,extension);
[17]       reduction = reduction << 1;
[18]       extension = extension >> 1;
[19]   }
[20]
[21]   printf("\n\t\tValeur initiale %d \t\tValeur initiale %8d\n",i,j);
[22]
[23] }
```

Lorsque l'on déplace d'un cran vers la gauche tous les bits qui composent un octet (concrètement, cela signifie qu'on ajoute un bit de valeur 0 à la suite des bits existants), l'opération équivaut à une multiplication par 2. Un déplacement vers la droite correspondrait à une amputation du bit de la valeur la plus élevée, c'est-à-dire à une division par 2. Etant donné que ces opérations ne consistent qu'en une translation de bits, elles sont sensiblement plus rapides que des opérations mathématiques.

L'opérateur Shift gauche "<< n" signale une translation de n bits vers la gauche, et l'opérateur Shift droite ">> n" symbolise une translation de même amplitude vers la droite.

Venons-en au programme. Dans les lignes 12 et 13 sont déclarées les variables entières *extension* et *reduction*, respectivement initialisées avec les valeurs 1 et 4000. Dans la boucle qui suit, répartie sur les lignes 14 à 19, la valeur la plus grande est divisée par 2 à chaque cycle, à l'aide de l'opérateur Shift droite, et la valeur la plus petite est multipliée par 2, par translation vers la gauche (Shift gauche). Les résultats de ces opérations sont ensuite renvoyés à l'écran.

Affichage produit par le programme SHIFT.C :

Shift gauche		Shift droite	
dec	hex	dec	hex
1	1	16384	4000
2	2	8192	2000
4	4	4096	1000
8	8	2048	800
16	10	1024	400
32	20	512	200
64	40	256	100
128	80	128	80

```
     256          100                64          40
     512          200                32          20
    1024          400                16          10
    2048          800                 8           8
    4096         1000                 4           4
    8192         2000                 2           2
   16384         4000                 1           1

Valeur initiale   1          Valeur initiale   16384
```

15.4. Accès individuel à des bits

Même si, jusque-là, vous avez pu modifier les bits d'un octet, il ne vous a pas encore été donné de les "voir". De plus, les seules modifications que vous avez pu effectuer sont celles obtenues à l'aide des opérateurs logiques agissant sur les bits dont nous disposions. Dans cette section, nous allons vous présenter une structure de données qui permet d'agir directement sur les bits et de les visualiser à l'écran. Le programme qui suit affiche à l'écran la configuration de bits correspondant à chaque touche actionnée au clavier, ainsi que le code ASCII correspondant, en notation décimale et hexadécimale.

Listing 75 : Le programme BITSTRUC.C

```
[1 ] /* Chapitre 15.4 */
[2 ] /* Nom du programme: BITSTRUC.C */
[3 ] /* Exemple d'accès direct à une configuration de bits */
[4 ]
[5 ] #include <dos.h>
[6 ]
[7 ] typedef struct                    /* Définir une structure de bits */
[8 ] {
[9 ]       unsigned bit_0 : 1;         /* chaque élément est un bit */
[10]       unsigned bit_1 : 1;
[11]       unsigned bit_2 : 1;
[12]       unsigned bit_3 : 1;
[13]       unsigned bit_4 : 1;
[14]       unsigned bit_5 : 1;
[15]       unsigned bit_6 : 1;
[16]       unsigned bit_7 : 1;
[17] } byte;                          /* 8 bits constituent 1 octet */
[18]
[19] union configuration_bits
[20] {
[21]   char c;              /* Variable de type char */
[22]   byte b;              /* Même variable que la configuration de bits */
[23] };
[24]
[25] union configuration_bits caractere;/*caractere_code est une variable */
```

```
[26]                                    /*variabledu type de données union*/
[27]                                    /* configuration_bits              */
[28] main()
[29] {
[30]
[31]
[32]   printf("Veuillez entrer une lettre -> ");
[33]   caractere.c = getch(); /* Chargement dans variable de type char */
[34]
[35]   /* Affichage sous forme de caractère ASCII et comme configuration
          de bits */
[36]
[37]   printf("\n\nLe caractère saisi est le caractère \t %c",
[38]          caractere.c);
[39]
[40]   printf("\nLa configuration de bits de ce caractère est la
          suivante : \t");
[41]   printf("%d ",caractere.b.bit_7); /* on commence par le 8ième bit */
[42]   printf("%d ",caractere.b.bit_6);
[43]   printf("%d ",caractere.b.bit_5);
[44]   printf("%d ",caractere.b.bit_4);
[45]   printf("%d ",caractere.b.bit_3);
[46]   printf("%d ",caractere.b.bit_2);
[47]   printf("%d ",caractere.b.bit_1);
[48]   printf("%d ",caractere.b.bit_0);
[49]   printf("\nCe qui équivaut au nombre hexadécimal :
          \t%x",caractere.c);
[50]   printf("\net en notation décimale à              :
          \t%d",caractere.c);
[51]
[52] }
```

Dans ce listing, la définition de la structure *byte* constitue une nouveauté. Elle est répartie sur les lignes 7 à 17, et se compose de huit variables sans signe relatif. La ligne

```
unsigned bit_0 : 1;
```

fixe à 1 bit la taille de la variable *bit_0*. La définition d'une structure *byte* constituée de huit éléments dont chacun a une taille de 1 bit instaure les conditions initiales requises pour gérer et manipuler un octet.

Dans ce but sont définies, dans l'union *configuration_bits* des lignes 19 à 23, une variable *b* du type *byte*, et une variable *c* du type *char*. Etant donné que ces deux types de données occupent le même espace mémoire, et qu'en raison de la déclaration sous forme d'union ils ont aussi la même adresse en mémoire de travail, il y a moyen d'accéder à cette structure aussi bien octet par octet que bit par bit.

Avant d'entrer dans le programme principal, nous voyons qu'en ligne 25 est déclarée une variable appelée *caractere*, dont le type de données est union *configuration_bits*. Il est dès lors possible d'accéder alternativement à la variable *caractere.c* du type *char*, ou à la configuration de bits *caractere.b*. C'est ainsi que, dans la ligne 33, un caractère quelconque est chargé depuis le clavier, puis retourné comme caractère de type *char*. Dans les lignes 41 à 48, le contenu de la même cellule de mémoire est à nouveau renvoyé, mais cette fois bit part bit, en commençant par le bit de poids le plus élevé.

Lancez une seconde fois le programme, et appuyez sur la touche <A>. Vous voyez apparaître les quatre types d'affichage possibles pour un signe : le caractère proprement dit, sa configuration de bits, identique d'ailleurs avec la notation binaire de son code ASCII, ainsi que son code ASCII exprimé en notation hexadécimale puis décimale.

Affichage renvoyé par le programme BITSTRUC.C :

```
Veuillez entrer une lettre ->
Le caractère saisi est le caractère                    A
La configuration de bits de ce caractère est la suivante : 0 1 0 0 0 0 0 1
Ce qui équivaut au nombre hexadécimal :                41
et en notation décimale à             :                65
```

15.5. Récapitulation

Les macros et fonctions de manipulation de caractères permettent d'établir une classification des caractères et de les manipuler. L'on peut ainsi déterminer pour chaque caractère s'il est du genre alphabétique *(isalpha())*, du genre numérique *(isdigit())*, ou d'un genre particulier *(isspace())*. Les caractères alphabétiques peuvent être transformés en minuscules à l'aide de *tolower()*, ou en majuscules à l'aide de *toupper()*.

Les opérateurs agissant sur des bits permettent d'effectuer des opérations logiques sur les bits. Les divers bits d'une variable peuvent, à cette occasion, être logiquement reliés par ET, OU, OU EXCLUSIF ou encore par NON.

Les opérateurs Shift gauche et droite permettent d'effectuer très rapidement des multiplications et des divisions. Shift gauche équivaut à une multiplication par 2, et Shift droite à une division par 2.

Grâce au double point (:) il est possible de définir des structures de données qui permettent d'accéder directement aux configurations de bits.

Exercices se rapportant au chapitre 15

Exercice 1 :
Ecrivez un programme qui permette d'ouvrir en lecture un fichier quelconque, et qui le visualise à l'écran. Une fonction

```
int touche_ascii(int c)
```

devra par la même occasion détecter et exclure de l'affichage les caractères non alphabétiques. Vous programmerez cette fonction de manière à ce qu'elle renvoie à la fonction appelante la valeur 0, pour tout caractère non alphabétique, et la valeur 1 dans le cas contraire.

Pour ce qui concerne la saisie du nom du fichier et l'ouverture de celui-ci, vous utiliserez les fonctions de l'exercice 1 du chapitre 12.

Programmez l'affichage des caractères en utilisant l'instruction suivante :

```
if (touche_ascii(c))
   putchar(c);
```

Exercice 2 :
Modifiez le programme précédent, de façon à ce que soient également retournés les caractères ASCII correspondant au retour chariot (Carriage Return), au saut de ligne (Line Feed), et à l'espace (Blank).

[redacted]

16 C, Pascal, BASIC - En quoi diffèrent-ils ?

A partir d'exemples de programmes, ce chapitre met en évidence les points communs et les différences des langages de programmation C, Pascal et BASIC. Nous en profiterons aussi pour décrire les causes d'erreurs auxquelles est sujet le néophyte...

L'objectif de ce chapitre est de faciliter le passage au langage C des habitués du Pascal ou du BASIC. Dans cette optique, nous mettrons en évidence les éléments communs, les différences et les sources d'erreurs possibles lors du passage de l'un des langages à l'autre. L'accent est essentiellement mis sur la comparaison du C et du Pascal. La "mise à l'écart" du BASIC est liée au fait que ce langage ne fait pas l'objet d'une normalisation universelle : la multiplicité des dialectes, qui dépendent souvent aussi du matériel utilisé, ne permet pas d'énoncer des règles générales. Il n'en demeure pas moins que, sur la base d'exemples, nous mettrons en évidence les différences fondamentales entre le C et le BASIC.

Le BASIC et le Pascal sont deux langages de programmation dont les philosophies sont très différentes, et dont l'usage est particulièrement répandu sur les PC. Le BASIC a acquis la réputation de langage de programmation non professionnel essentiellement destiné à l'initiation de celui qui veut se lancer dans la programmation. Il ne requiert aucune véritable connaissance en informatique, et le programmeur n'a pas besoin de tenir compte de la structure de ses données. Les plus anciens dialectes du basic ne connaissaient pas le découpage en programme principal et sous-programmes. Le nombre des opérateurs disponibles était réduit, et la distinction entre variables locales et

globales était inexistante. Les dialectes plus modernes du BASIC proposent par contre les structures de programmation les plus importantes, mais restent extrêmement "flous" lorsqu'il s'agit de programmation proche du système.

A la différence du BASIC, le Pascal a été conçu pour former les spécialistes en informatique. Il est fortement structuré et requiert une technique de programmation très disciplinée, dans laquelle les structures des données et du programme sont bien séparées. Tout programme Pascal se compose d'un programme principal et d'un nombre plus ou moins important de sous-programmes. Le Pascal connaît une multitude d'opérateurs, et fait la distinction entre variables locales et globales.

La comparaison entre le C et le Pascal est d'autant plus facile à mener que les deux langages bénéficient d'un haut niveau de normalisation. C'est la raison pour laquelle nous aborderons de manière approfondie les structures de données et de contrôle, dans la section 16.2. Pour finir, dans la section 16.3, nous décrirons les erreurs de programmation les plus fréquentes qui se produisent lorsqu'un programmeur passe d'un autre langage au C.

16.1. Différences fondamentales avec le BASIC

Le BASIC compte, aux yeux de nombreux programmeurs, comme étant le "langage du débutant", facile à apprendre, mais incitant à évoluer vers une programmation chaotique, et inadapté pour la programmation de tâches complexes. Les programmes en BASIC que nous présentons ci-après fonctionnent dans tout compilateur BASIC, et peuvent aussi être traités dans l'interpréteur GWBASIC, livré avec le système d'exploitation, ou sous QBASIC (fourni avec MS-DOS 5.0).

A l'aide du programme BASIC HELLO.BAS et du programme C HELLO3.C, nous mettrons en évidence les différences essentielles entre le BASIC et le C. Par la même occasion, nous verrons toutefois aussi que ces deux langages présentent des points communs.

Listing 76: Le programme BASIC HELLO.BAS

```
100 CLS
200 A$ = "Bonjour tout le monde"
300 PRINT A$
400 B$ = A$
500 PRINT B$
```

Listing 77 : Pour comparaison : Le programme HELLO3.C

```
[1 ]    main()
[2 ]    {
[3 ]        char *a;
[4 ]        char *b;
[5 ]        a = (char*) malloc(21); b = (char*) malloc(21);
[6 ]        a = "Bonjour tout le monde";
[7 ]        printf("%s",a);
[8 ]        b = a;
[9 ]        printf("%s",b);
[10]    }
```

Trois différences par rapport au C sautent aux yeux. Dans les lignes 200 et 400 du programme BASIC, des valeurs sont attribuées à des variables qui n'ont pas été déclarées. La variable est en fait déclarée alors que lui est attribuée une valeur. Ceci n'est pas possible en langage C, où il faut d'abord déclarer la variable. La deuxième différence visible réside dans le point virgule qui doit achever une instruction en C, et qui n'existe pas en BASIC où aucun symbole de fin d'instruction n'est prévu. Les instructions y sont définies par un numéro de ligne, et conclues par une pression sur <Return>. De plus, l'exemple de programme BASIC ci-dessus montre que le BASIC ne connaît pas la notion de programme principal, et par conséquent pas de structure de bloc. Un programme BASIC commence par l'instruction de numéro le moins élevé, et s'achève par la dernière instruction ou par le mot clé END.

Les listings précédents mettent toutefois en évidence un point commun : l'opérateur d'affectation "=", qui attribue à la variable située à sa gauche la valeur de la variable qui se trouve à sa droite.

Listing 78: Le programme BASIC RELATION.BAS

```
100 CLS
200 INPUT "Veuillez entrer un nombre ", A
300 INPUT "Veuillez entrer un deuxième nombre ", B
400 IF A = B THEN PRINT "Les deux nombres sont égaux "
            ELSE PRINT "Les deux nombres sont différents "
```

Le BASIC utilise le même symbole pour comparer des variables et pour affecter une valeur à une variable. Si la relation établie au moyen du signe d'égalité suit le mot clé *IF*, le BASIC interprète cette relation comme comparaison, une opération logique qui renvoie l'une des valeurs *vrai* ou *faux*. Une confusion entre comparaison et affectation de valeur n'est pas possible. Il en va tout différemment en C. Les conséquences d'une telle confusion en C sont mises en évidence par l'exemple de programme suivant.

Listing 79: Le programme INFINI.C

```
[1 ] /* Nom du programme : INFINI.C */
[2 ] /* Exemple de boucle infinie involontaire en C */
[3 ]
[4 ] main()
[5 ] {
[6 ]    int compteur;  /* Variable incrémentée  */
[7 ]
[8 ]    compteur = 0;
[9 ]
[10]    while (compteur < 6)
[11]    {
[12]      printf("La valeur de la variable incrémentée est %d\n]",compteur);
[13]      compteur = compteur + 1;
[14]      if (compteur = 5)
[15]         printf("La boucle va s'arrêter incessamment\n");
[16]    }
[17]
[18] }
```

Dans ce programme de démonstration, la ligne 14 fait en sorte que la valeur 5 soit en permanence transmise à la variable *compteur*. La condition d'arrêt de la boucle ne pourra donc jamais être atteinte, et cette boucle sera parcourue à l'infini.

Le BASIC appartient à la catégorie des langages dits non structurés. Cela signifie que rien est imposé au programmeur pour ce qui concerne la structure d'un programme. Une variable peut par exemple être déclarée par affectation d'une valeur à un emplacement quelconque du programme. La commande GOTO permet d'établir un renvoi inconditionnel vers toute instruction depuis n'importe quelle ligne. Etant donné que le BASIC ne connaît pas la subdivision en programmes et sous-programmes, la structure d'un programme résulte de renvois suivis de retours au point initial du renvoi. Voyons également un exemple qui repose sur ce principe :

Listing 80: Le programme TAXVALAJ.BAS

```
50 REM Calcul d'un montant TVA incluse
100 CLS
150 TVA = 1.186
200 INPUT "Veuillez entrer un montant hors taxes ", NET
300 GOTO 1000
400 PRINT "Lorsque le taux de la TVA est de " TVA
500 PRINT "le montant toutes taxes est de " BRUT
600 END
1000 REM C'est ici qu'est calculé le montant avec TVA
1100 BRUT = NET * TVA
1200 REM puis retour
```

```
1300 GOTO 400
```

Dans la ligne 300 est programmé ici un renvoi sur la ligne 1000. Dans cette dernière est calculé le montant toutes taxes, mémorisé dans la variable *brut*. Le programme revient ensuite dans la ligne qui suit l'instruction de renvoi, soit, dans notre exemple, la ligne 400. Le renvoi des résultats est programmé dans les lignes 400 et 500, et la ligne 600 marque la fin du programme.

Deux particularités méritent de retenir l'attention dans le programme précédent. La première concerne la variable *brut*, qui est globale : elle n'est déclarée que dans la ligne 1100, et peut malgré tout être utilisée en ligne 500 pour renvoyer le résultat déterminé par le programme. L'utilisation d'une telle technique est impossible en C. La seconde particularité concerne la ligne 400, dans laquelle il est nécessaire de revenir : en l'absence d'ordre de renvoi en ligne 1300, les lignes 400 à 600 ne seraient pas exécutées. Le programme s'achèverait à la suite du traitement de la ligne 1300.

Si maintenant vous observez le programme C équivalent, vous constatez que la conception de celui-ci est plus claire. Dans ce programme, il faut de plus transmettre le montant net (hors taxes) à la fonction *tva()*, qui retourne ensuite la valeur calculée appropriée à la fonction appelante (soit ici *main()*). L'exemple met toutefois aussi en évidence que la programmation de la saisie est plus facile à réaliser en BASIC qu'en C. En BASIC, l'invitation à la saisie et la mémorisation dans une variable peuvent être effectuées en une seule étape. En C, deux activations de fonctions distinctes sont nécessaires. C'est en effet au programmeur qu'il appartient de faire en sorte que l'adresse de la variable à charger soit transmise à la fonction d'extraction *scanf()*.

Listing 81: Le programme TAXVALAJ.C

```
[1 ] /* Nom du programme : TAXVALAJ.C */
[2 ] /* Structuration d'un programme par activation de la fonction */
[3 ]
[4 ] #define TVA 1.186          /* Facteur du calcul du montant TVA incluse */
[5 ]
[6 ] float tva(float valeur)  /* Ceci est une fonction */
[7 ] {
[8 ]   return( (valeur * TVA) );
[9 ] } /* en fin de fonction, retour au programme principal */
[10]
[11] main()
[12] { /* c'est ici que commence le programme principal */
[13]   float net,        /* Valeur saisie   */
[14]   brut;             /* Valeur calculée */
[15]
```

```
[16]    printf("Veuillez entrer un montant hors taxes ");
[17]    scanf("%f", &net); /* Montant hors taxes sous forme de nombre */
[18]                       /* à virgule flottante (float)             */
[19]    brut = tva(net);   /* Le montant TTC (brut) est le résultat   */
[20]                       /* renvoyé par la fonction tva()           */
[21]
[22]    printf("Lorsque le montant hors taxes est de %-6.2f ",net);
[23]    printf("le prix TTC à payer est de %-8.2f",brut);/* Affichage */
[24] } /* Fin du programme principal */
```

En C aussi, il est possible de déclarer des variables à des endroits quelconques d'un programme, mais à condition que la déclaration soit effectuée à l'intérieur d'un bloc du programme. Cette possibilité, à laquelle s'ajoute le fait que l'instruction GOTO est incluse au langage, pousse les détracteurs du C à ranger également ce langage dans la catégorie des langages non structurés qui peuvent inciter le programmeur à un manque de rigueur dans la rédaction des programmes, de manière à ce que ceux-ci ne puissent pas être suivis facilement par un tiers.

Mais à la différence du BASIC, il existe en C une distinction très stricte entre variables locales et variables globales. Les variables d'un bloc de programme C ne sont valables que dans le bloc dans lequel elles on été déclarées. En-dehors de ce bloc, elles n'existent pas. Ce n'est que cette facette du langage qui permet une construction modulaire conséquente de programmes. En BASIC, par contre, toutes les variables sont globales.

16.2. Différences fondamentales avec le Pascal

Dans les sections qui suivent, vous découvrirez les différences essentielles entre le Pascal et le C. Les programmes que nous vous présenterons utilisent des mots clés, des procédures et des fonctions qui appartiennent au Pascal dans sa version originale, et dont la compilation peut donc être réalisée à l'aide de la plupart des compilateurs.

16.2.1. Types de données et opérateurs

Bien que les deux langages présentent un grand nombre de similitudes, il existe une différence fondamentale concernant en particulier les structures de données et de contrôle qui y sont implémentées, et les opérateurs disponibles. Les programmes DATA.PAS et DATA.C mettent en évidence les différences entre

types de données et opérateurs. Les deux programmes possèdent la même structure, et produisent le même affichage.

Listing 8 2 : Le programme DATA.PAS

```
[ 1] program data_pas;
[ 2] { ********************************************************* }
[ 3] { * Ce programme montre les types de données disponibles   * }
[ 4] { * en Pascal, et met en évidence les différences essen-   * }
[ 5] { * tielles par rapport au C pour ce qui touche aux struc- * }
[ 6] { * tures de données et aux opérateurs utilisables.        * }
[ 7] { ********************************************************* }
[ 8]
[ 9]   const
[10]   MAX = 10;  { La constante symbolique MAX reçoit la valeur 10 }
[11]
[12]   var        { Début de la déclaration des Variables }
[13]     nombre1,
[14]     nombre2  :integer ; { Equivaut au type de données int en C }
[15]     champint : array [1..MAX] of integer; {Tableau d'entiers avec 10
                                                éléments }
[16]     caractascii : char;      { équivaut au même type de données en C  }
[17]     caractere   : byte;      { équivaut au type unsigned char en C    }
[18]     nombre3     : real;      { équivaut au type de données float en C }
[19]     vrai        : Boolean;   { vrai est un type de données logique    }
[20]     texte       : string[80]; { texte est une chaîne dont la longueur }
[21]                               { est 80 caractères                    }
[22]   begin  { programme principal }
[23]
[24]    texte := 'Ceci est une chaîne en Pascal .';  { Affectation
                                                       directe de chaîne }
[25]    writeln(texte);          { Routine d'affichage en Pascal      }
[26]    writeln(texte[1]);       { Renvoi du 1er caractère du tableau }
[27]    vrai := true;            { Attribuer une valeur de vérité     }
[28]    if vrai then             { puis l'utiliser                    }
[29]     begin
[30]      writeln('Cet affichage résulte du fait que vrai est "vrai" ');
[31]     end;
[32]
[33]     nombre1 := 1;
[34]     while vrai do   { Tant que vrai est "vrai" }
[35]     begin
[36]      champint[nombre1] := nombre1;     { commence avec l'index le plus
                                              faible }
[37]          write(champint[nombre1],' '); { Affichage du contenu
                                              du champ }
[38]          nombre1 := nombre1 + 1;       { Incrémentation de nombre1
                                              de la valeur 1 }
[39]          if nombre1 > MAX then
[40]            begin
[41]             vrai := false;             { à présent, vrai est "faux " }
[42]            end;
[43]        end;
[44]
```

```
[45]        writeln; { Ligne vide }
[46]        writeln('Tableau d''entiers initialisé et retourné ');
[47]        writeln('et maintenant reprenons');
[48]        writeln;
[49]
[50]        nombre1 := MAX;
[51]        while nombre1 > 0 do { Tant que l'indice est valable }
[52]          begin
[53]            champint[nombre1] := nombre1; { commence par l'indice le
                                                plus élevé }
[54]            write(champint[nombre1],' '); { Affichage du contenu du
                                                champ }
[55]            nombre1 := pred(nombre1);     { nombre1 est décrémenté de 1-
                                                Opérateur de décrémentation-}
[56]          end;
[57]
[58]        caractascii := 'A';    { Attribution d'une constante littérale }
[59]        write(caractascii);
[60]        caractascii := succ(caractascii);{Opérateur d'incrémentation }
[61]        write(caractascii);    {Retourner le caractère ASCII
[62]                                suivant}
[63]
[64]    end. { C'est ici que s'achève le programme }
```

Listing 83: Le programme DATA.C

```
[ 1]    /* DATA.C */
[ 2]    /* Programme de démonstration: Comparaison des langages C  */
[ 3]    /* et Pascal équivaut au programme DATA.PAS */
[ 4]
[ 5]    #define MAX 10   /* équivaut à const en Pascal */
[ 6]
[ 7]    int nombre1,
[ 8]        nombre2,
[ 9]        champint[10];
[10]     float nombre3;
[11]     char caractascii;
[12]     char texte[81]; /* La longueur maximale du texte est de 80! */
[13]                     /* caractères                               */
[14]     main()
[15]     { /* Début du programme principal */
[16]
[17]        strcpy(texte,"Ceci est une chaîne en C.......");
[18]                /* Affectation directe impossible ! */
[19]        printf("\n%s\n",texte);
[20]       printf("%c\n",texte[0]);     /* Renvoyer le premier(!) */
[21]                                    /* caractère de "texte"   */
[22]
[23]        if (nombre1 = 1) /* Affectation à l'aide de l'opérateur = */
[24]                         /* dans une instruction unique           */
[25]         {
[26]          printf("Cet affichage résulte du fait que nombre1 est
                 \"vrai\" = 1 \n\n ");
[27]         }
```

```
[28]        nombre2 = 0;
[29]        while (nombre1)
[30]          {
[31]          champint[nombre2] = nombre2;
[32]          printf("%d", champint[nombre2]);
[33]          nombre2 = nombre2 + 1;
[34]          if ( (nombre2 = nombre2 + 1) > 9) /* L'indice du dernier
                                                   élément est 9! */
[35]            {
[36]            nombre1 = 0;                    /* à présent nombre1 est
                                                   "faux" */
[37]            }
[38]          }
[39]          printf("\nChamp entier initialisé et retourné\n");
[40]          printf("\Revenons au départ\n");
[41]
[42]          nombre2 = MAX;
[43]          while (nombre2 > -1)
[44]          {
[45]           printf("%d ",champint[--nombre2]); /* décrémenter d'abord, */
[46]           }                                  /* utiliser ensuite     */
[47]           caractascii = 'A';
[48]           printf("\n%c ", caractascii++);    /* Incrémenter          */
[49]           printf("%c", caractascii);         /* après utilisation    */
[50]    }
```

A la différence du Pascal, le C ne connaît pas de type de données logique dont la valeur intrinsèque peut être vrai (en anglais : true) ou faux (en anglais : false). En C, est considérée comme vraie toute variable entière dont la valeur est non nulle, et fausse celles dont la valeur est 0. Alors qu'en C des types de données entiers sans signe relatif peuvent être déclarés à l'aide du mot clé *unsigned*, de telles données sont inconnues du Pascal. Le type de données *byte* du Pascal équivaut au type *unsigned char* du C. La déclaration de constantes est introduite, en Pascal, par le mot clé *const* qui correspond à la déclaration *#define* en C.

La plus grande différence concerne toutefois la déclaration et le traitement des tableaux et des chaînes. Reportez-vous pour cette comparaison aux lignes 15 et 20 du programme Pascal, et aux lignes 9 et 12 du programme C.

```
[15]      champint : array [1..MAX] of integer;
[20]      texte    : string[80];
```

```
[ 9]        champint[10];
[12]      char texte[81];
```

Alors qu'en C la valeur de l'indice le plus petit est 0, la numérotation des tableaux en Pascal commence à 1. Cette différence se reporte évidemment sur les numéros plus élevés des éléments. En Pascal, la longueur d'un tableau est définie

par la spécification du premier et du dernier élément. La déclaration en C s'effectue par spécification du type de données, du nom du tableau et du nombre de ses éléments. Le C réserve alors la plage de mémoire nécessaire, en fonction du nombre d'éléments et du type de données, et affecte au nom du tableau l'adresse de l'origine de la plage de mémoire réservée.

Des différences sont aussi à noter pour ce qui touche au traitement des chaînes (strings). Bien que le type de données *string* ne soit pas prévu dans le Pascal standard, la plupart des compilateurs de Pascal connaissent ce type. Il équivaut à un tableau de données du type *char*. La différence essentielle réside dans le fait que, comme vous pouvez le constater dans la ligne 24 du programme Pascal, des chaînes alphanumériques constantes peuvent être affectées directement à des variables. Ceci n'est pas possible en C.

En C, la déclaration d'une chaîne peut être réalisée alternativement comme tableau du type de données *char*, ou comme pointeur dirigé sur le type de données *char*. Comme vous pourrez le constater dans la section 16.3 qui suit, les deux procédés s'accompagnent de conséquences (il faudrait presque parler d'effets secondaires) qui, si vous n'en tenez pas compte, engendrent des erreurs de programmation.

16.2.2. Opérateurs d'affectations

Les deux langages sont très différents pour ce qui touche à l'attribution ou à la comparaison de valeurs. En Pascal, l'attribution d'une valeur au moyen de l'opérateur := est une instruction qui doit impérativement être conclue par le point virgule ou le symbole de fin de bloc. En C, l'affectation est un opérateur qui peut être employé dans une expression quelconque. Vous pouvez bien voir cette différence en observant les lignes 25 à 50 du programme Pascal, et la ligne 34 du programme C. Alors qu'en C il suffit d'une ligne d'instructions pour l'affectation suivie de la comparaison d'une variable avec une autre variable ou une constante, en Pascal l'affectation doit être réalisée d'abord. Ce n'est qu'ensuite que la nouvelle valeur de la variable pourra être comparée à une autre.

16.2.3. Structures de contrôle

Dans cette section, nous découvrirons les différences fondamentales de programmation des structures de contrôle, à l'aide des programmes CASE.C(PAS), BOUCLE.C(PAS) et FONCTION.C(PAS). Nous nous baserons d'abord sur les deux programmes qui agissent par différenciation de cas. Ils vérifient si un nombre saisi est supérieur ou inférieur à 5.

Listing 84: Le programme CASE.PAS

```
[1 ]  program structure_case_pas;
[2 ]  { ******************************************************************* }
[3 ]  { * Ce programme Pascal est une démonstration de la structure CASE * }
[4 ]  { ******************************************************************* }
[5 ]
[6 ]   const
[7 ]       MILIEU = 5;
[8 ]
[9 ]   var
[10]       nombre : integer;
[11]
[12]   begin
[13]
[14]       write('Veuillez entrer un nombre inférieur à 11 ');
[15]       readln(nombre);
[16]       case nombre of
[17]           1..4 : writeln('Le nombre entré est inférieur à ', MILIEU);
[18]           5    : writeln('Le nombre entré est égal à ',MILIEU);
[19]           else
[20]               writeln('Le nombre entré est supérieur à ',MILIEU);
[21]       end;
[22]   end.
```

Comme vous pouvez le constater en ligne 17, le Pascal permet d'utiliser des intervalles de valeurs dans la structure *case*. Il est possible de programmer ici une instruction pour le cas où la variable *nombre* prendrait une valeur comprise entre 1 et 4 (inclus). Ceci n'est pas possible en C. Dans ce langage, l'on ne peut faire appel qu'à une valeur entière unique pour engager une ramification. Une seconde différence déterminante est fréquemment source d'erreur chez le néophyte qui a déjà programmé en Pascal : en Pascal, les instructions qui suivent une structure *case* sont automatiquement non exécutées ("sautées"). Ainsi, lorsque *nombre* prend l'une des valeurs présentées, seule l'instruction qui suit le double point sera exécutée, et toutes celles qui suivent ne sont pas prises en compte. L'exemple suivant montre qu'en C, il faut programmer une interruption par *break* pour obtenir le même résultat.

Listing 85: Le programme CASE.C

```
[1 ] /* CASE.C /*
[2 ] /* Programme de démonstration de la structure SWITCH en C. Le pro- */
[3 ] /* gramme montre la différence avec la structure CASE du Pascal    */
[4 ]
[5 ] #define MILIEU 5
[6 ]
[7 ] int nombre;
[8 ]
[9 ] main()
[10] {
[11]    printf("Veuillez entrer un nombre inférieur à 11 -> ");
[12]    scanf("%d", &nombre);
[13]    switch (nombre)
[14]    { /* Début de switch */
[15]      case 1:
[16]      case 2:
[17]      case 3:
[18]      case 4:{
[19]                printf("Le nombre saisi est inférieur à %d", MILIEU);
[20]                break; /* vous quittez ici la structure switch */
[21]              }
[22]      case 5:{
[23]                printf("Le nombre saisi est égal à %d", MILIEU);
[24]                break;
[25]              }
[26]    default:
[27]    printf("Le nombre saisi est supérieur à %d", MILIEU);
[28]    } /* Fin de switch */
[29] }
```

C'est la raison pour laquelle les instructions des lignes 15 à 20 du programme C sont équivalentes à l'instruction de la ligne 17 du programme Pascal. Etant donné qu'aucun "*break*" n'a été programmé avant la ligne 20, l'instruction

```
printf("Le nombre saisi est inférieur à %d", MILIEU);
```

s'applique à toutes les valeurs comprises entre 1 et 4 (inclus). Ce n'est qu'en raison de l'interruption programmée au moyen de *break* en ligne 24 que les lignes 26 et 27 ne sont plus traitées.

Les deux langages ont en commun la possibilité de programmer par l'intermédiaire de *else* (respectivement *default*) des instructions qui ne seront exécutées que si aucune des valeurs présentées n'est égale à celle de la variable testée.

Les deux exemples qui suivent présentent la comparaison des structures de boucles implémentées dans les deux langages. Le Pascal et le C connaissent tous deux les boucles testées en entrée et celles testées en sortie.

Listing 86: Le programme BOUCLE.PAS

```
[1 ] program structures_de_boucles_en_pascal;
[2 ] { ******************************************************** }
[3 ] { *Ce programme Pascal présente les structures de boucles * }
[4 ] { *implémentées en Pascal, et équivaut au programme C du  * }
[5 ] { *même nom                                               * }
[6 ] { ******************************************************** }
[7 ] uses crt;
[8 ]
[9 ] const
[10]   ARRET = '9';
[11]
[12] var
[13]   caractere : char;
[14]   help    : integer;    { sera nécessaire pour interrompre }
[15]                         { la boucle FOR }
[16]
[17] begin
[19]
[20]    caractere := 'x';
[21]    writeln('Première boucle - Appuyer sur une touche -
                Arrêt par pression sur 9 ');
[22]    while (caractere <> '9') do
[23]    begin
[24]        caractere := readkey;
[25]        write(caractere);
[26]    end;
[27]     writeln;
[28]
[29]     caractere := 'x';
[30]
[31]     writeln('Deuxième boucle - Appuyer sur une touche -
                 Arrêt par pression sur 9 ');
[32]     for help := 0 to 9999 do
[33]     begin
[34]       caractere := readkey;
[35]       write(caractere);
[36]       if caractere = '9' then help := 9999;
[37]    end;
[38]
[39]    writeln;
[40]    caractere := 'x';
[41]
[42]    writeln('Troisième boucle - Appuyer sur une touche -
                Arrêt par pression sur 9 ');
[43]    repeat
[44]        caractere := readkey;
[45]        write(caractere);
[46]    until (caractere = ARRET);
[47]
[48] end.
```

Listing 87: Le programme BOUCLE.C

```
[1 ] /* BOUCLE.C */
[2 ] /* Programme de démonstration des structures de boucles en C. */
[3 ] /* Le programme est identique au programme Pascal BOUCLE.PAS */
[4 ] #include <stdio.h>
[5 ]
[6 ] #define ARRET '9'
[7 ]
[8 ] char caractere;
[9 ]
[10] main()
[11] {
[12]  caractere = 'x';
[13]  printf("Première boucle - Appuyer sur une touche -
            Arrêt par pression sur 9 \n ");
[14]  while( caractere != ARRET)     /* Identique en C et */
[15]  {                              /* en Pascal */
[16]    caractere = getch();
[17]    putchar(caractere);
[18]  }
[19]  caractere = 'x';
[20]  printf("\nDeuxième boucle - Appuyer sur une touche -
             Arrêt par pression sur 9 \n ");
[21]  for (; caractere != ARRET; caractere = getch(),
          putchar(caractere));
[22]       /* Toutes les opérations sont incluses dans la boucle FOR, */
[23]       /* ce qui est impossible en Pascal */
[24]
[25]  caractere = 'x';
[26]  printf("\nEt voici la troisième boucle -
            Arrêt par pression sur 9\n ");
[27]  do            /* équivaut de par sa structure */
[28]  {             /* à REPEAT .. UNTIL */
[29]     caractere = getch();
[30]     putchar(caractere);
[31]  } while ( caractere != ARRET);
[32]
[33] }
```

En Pascal, les boucles testées en entrée sont programmées à l'aide de *while..do*, et correspondent ainsi aux boucles *while* du C. La boucle testée en sortie est programmée à l'aide de *repeat..until* en Pascal, et par *do..while* en C. La différence essentielle réside dans le fait que la boucle Pascal est parcourue jusqu'à ce que soit remplie une condition qui suit *until*, comme dans

```
until (caractere = ARRET);
```

alors qu'en C la boucle est parcourue tant que la condition qui suit *while* n'est pas remplie. C'est pourquoi, dans la ligne 31, il nous faut programmer ici :

```
while ( caractere != ARRET);
```

La boucle particulièrement compacte de la ligne 21

```
for (; caractere != ARRET; caractere = getch(),putchar(caractere));
```

ne peut pas être programmée sous cette forme en Pascal. Dans le programme Pascal, il y a aussi une boucle *for* (lignes 32 à 37) qui effectue le même travail que la boucle C ci-dessus. Il a toutefois fallu programmer une valeur d'interruption pour la boucle de comptage, qui garantisse qu'un grand nombre de saisies pourront être effectuées au clavier. Dès que l'utilisateur actionne la touche <9>, la valeur 9999 est attribuée à la variable de boucle, et l'exécution de celle-ci s'achève.

Les deux programmes montrent aussi qu'il existe un point commun important aux deux langages : le Pascal et le C sont des langages orientés bloc. Chaque corps de boucle est parfaitement délimité par les délimiteurs de bloc *begin..end* (respectivement {..}), et traité en conséquence. La structure *repeat..until* fait exception à cette convention : les deux mots clés font eux-mêmes office de délimiteurs de bloc.

Pour finir ce chapitre, nous allons encore aborder les différences qui concernent la programmation de sous-programmes, de fonctions et de procédures. Les deux langages de programmation supportent la programmation modulaire, et gèrent le fonctionnement d'un programme à partir d'un programme principal. Les programmes de démonstration qui suivent nous montreront que le Pascal fait la distinction entre deux sortes de sous-programmes. En Pascal, les sous-programmes qui ne renvoient aucune valeur au programme principal sont appelés procédures, et sont programmés à l'aide du mot clé *procedure*. Les sous-programmes qui renvoient une valeur sont appelés fonctions, et sont programmés à l'aide du mot clé *function*. Le programme FONCTION.PAS montre, dans ses lignes 17 à 20, la programmation d'une procédure, et dans les lignes 22 à 36 celle d'une fonction. Ce programme détermine la valeur la plus grande dans un ensemble de 5 nombres, et la renvoie à l'écran. La visualisation est programmée à l'aide de la procédure *affichage*, et la détection de la plus grande valeur à l'aide de la fonction *nombremaxi*.

Listing 88: Le programme FONCTION.PAS

```
[ 1] Program fonction_et_procedure_en_pascal;
[ 2] {*********************************************************}
[ 3] { * Ce programme Pascal montre la différence entre        * }
[ 4] { * fonctions et procédures, et est identique au programme * }
[ 5] { * C de même nom.                                        * }
[ 6] {*********************************************************}
[ 7]
[ 8] const
[ 9]   MIN = 1;
[10]   MAX = 5;
[11]
[12] var
[13]   i,       { Variable de boucle }
[14]   maximum      : integer;
[15]   intarray : array [MIN..MAX] of integer;
[16]
[17]   procedure affichage (nombre : integer); { une procédure n'a pas de
                                                 résultat }
[18]   begin
[19]      writeln('Le plus grand nombre entré est ',nombre);
[20]   end;
[21]
[22]   function nombremaxi :Integer; { La fonction retourne un nombre
                                       entier en guise de résultat }
[23]   var
[24]      index,
[25]      help    : integer;
[26]
[27]   begin
[28]      help := -999;
[29]
[30]      for index := MIN to MAX do
[31]      begin
[32]         if intarray[index] > help then
[33]         help := intarray[index];
[34]      end;
[35]      nombremaxi := help; { Le résultat de la fonction nombremaxi
[36]   end;                   { est le contenu de  la variable help }
[37]
[38]   begin  { c'est ici que commence le programme principal }
[39]
[40]      i := MIN;
[41]      writeln('Veuillez entrer un nombre supérieur à -999 et
                 inférieur à 999 ');
[42]      while ( i <= MAX) do  { Chargement de 5 valeurs dans le tableau
                                  d'entiers }
[43]         begin
[44]         read(intarray[i]);
[45]         i := succ(i);
[46]      end;
[47]      maximum := nombremaxi;    { maximum est égal à la valeur
                                      renvoyée par la fonction nombremaxi }
```

```
[48]        affichage(maximum);      { maximum est la valeur transmise à la
                                       procédure affichage }
[49] end.
```

La ligne 47

```
maximum := nombremaxi;
```

met en évidence une différence fondamentale par rapport au C : vous constatez qu'en Pascal il n'est pas nécessaire de mentionner les parenthèses d'une fonction lorsqu'aucun argument n'est attribué à cette fonction. La ligne 35

```
nombremaxi := help;
```

montre une autre différence : en Pascal, une valeur est attribuée à la fonction elle-même. En C, c'est la fonction qui retourne une valeur.

Listing 89: Le programme FONCTION.C

```
[1 ] /* FONCTION.C */
[2 ] /* Programme de démonstration de fonctions avec et sans  */
[3 ] /* renvoi de valeur. Identique à FONCTION.PAS */
[4 ]
[5 ] #include <stdio.h>
[6 ]
[7 ] void affichage(int maximum)
[8 ] {
[9 ]   printf("\nLe plus grand nombre saisi est %d ", maximum);
[10] }
[11]
[12] int nombremaxi(int intarray[])
[13] {
[14]   int i,
[15]   help = -999;    /* Champ d'aide au tri */
[16]
[17]   for (i=0; i< 5;i++)
[18]   {
[19]     if (intarray[i] > help)
[20]       help = intarray[i];
[21]   }
[22]   return(help);
[23] }
[24]
[25] main()
[26] {
[27]   int nombre[5],
[28]       i,
[29]       max;
[30]
[31]
[32]   printf("Veuillez entrer 5 nombres entiers quelconques \n");
```

```
[33]    printf("compris entre -999 et +999 \n");
[34]    for (i=0; i< 5; printf("\nSaisie -> "),
[35]          scanf("%d", &nombre[i]),i++);
[36]
[37]    max = nombremaxi(nombre);
[38]    affichage(max);
[39]    getch();
[40] }
```

16.3. Erreurs de programmation fréquentes

Dans ce chapitre, nous décrirons les erreurs de programmation les plus fréquentes auxquelles sont essentiellement sujets les programmeurs "qui viennent" du BASIC ou du Pascal. Conformément à ce qu'ont montré les programmes précédents, ces erreurs sont le plus souvent liées aux différences de notations symboliques ou de types de données dans les différents langages.

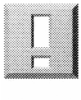
Confusion entre opérateur de comparaison et opérateur d'affectation

Pour les programmeurs qui viennent de quitter le Pascal, cette erreur risque d'être la plus fréquente. Le plus gênant, c'est que dans ce cas de figure il n'y a pas d'erreur de syntaxe. Il en résulte des erreurs logiques de programmation dont l'origine est extrêmement difficile à localiser dans les programmes volumineux.

Non respect de l'écriture majuscule / minuscule

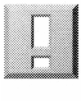
En tant que néophyte, vous ne devrez pas perdre des yeux qu'en C la variable *Index* n'est pas la même que la variable *index*. De telles erreurs engendrent généralement une erreur de syntaxe qu'il est possible de corriger durant la compilation.

Apostrophes à la place des guillemets



Cette erreur se produit le plus souvent lors de l'utilisation de *printf()*.

Mauvaise indexation de tableau



En C, le premier indice d'un tableau est 0. L'erreur la plus répandue dans ce domaine est celle commise par le néophyte qui utilise un indice trop élevé (l'indice 10, par exemple, pour un tableau de dix éléments).

Pointeur non initialisé

Cette erreur pourrait être la plus grave. Le danger, dans ce cas, est que les compilateurs afficheront probablement un message d'erreur, mais que l'exécution des programmes se poursuivra malgré tout. Voyons cela sur un exemple extrait d'un programme :

```
int *pointer;
*pointer = 400
printf("Le pointeur est dirigé sur la valeur %d\n", *pointer);
```

Comme nous n'affectons aucune adresse au pointeur *pointer*, celui-ci est dirigé sur une adresse de la mémoire totalement aléatoire. Si celle-ci est, par hasard, l'adresse d'une fonction du programme, celle-ci sera écrasée par la valeur 400 (deuxième ligne du programme), et le programme "se plantera".

Plus un programme est volumineux, plus la probabilité d'un plantage lié à la non initialisation d'un pointeur est grande. Il est impératif qu'un pointeur soit toujours initialisé.

Confusion entre tableaux de type *char* et pointeurs dirigés sur *char*

A des pointeurs dirigés sur le type de données *char*, vous pouvez attribuer directement des chaînes de caractères. Il est impératif d'avoir préalablement réservé l'espace mémoire nécessaire à l'aide des fonctions *malloc()* (respectivement *calloc()*). En l'absence de cette précaution, des plantages de programmes sont aussi à craindre dans ce cas. Dans le pire des cas, c'est l'ordinateur lui-même qui se bloque.

Lorsque vous travaillez avec des tableaux, les caractères doivent être copiés dans le tableau, à l'aide de la fonction *strcpy()* par exemple. Dans ce cas, une affectation directe par l'intermédiaire de l'opérateur d'affectation n'est pas possible.

Transmission d'une valeur à la place d'une adresse

Cette erreur se produit le plus souvent lors de l'utilisation de la fonction *scanf()*. Pour toutes les variables qui ne sont pas des pointeurs, il faut spécifier dans ce cas l'opérateur d'adressage. L'extrait de programme suivant rappelle cette technique :

```
int nombre;
char *string
string = malloc(10);    /* Initialisation du pointeur */
scanf("%d",&nombre);    /* Variable entière, d'où opérateur d'adressage */
scanf("%s",string);     /* string est une adresse, d'où pas d'opérateur
                           d'adressage */
```

17 Annexes

17.1. Les mots clés du C

Les termes repérés par AN sont des extensions répondant à la norme ANSI.

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
const (AN)	if	typedef
continue	int	union
default	long	unsigned (AN)
do	register	void (AN)
double	return	volatile (AN)
else	short	while
enum (AN)	signed (AN)	

17.2. Les opérateurs en C

Opérateurs mathématiques et affectation

Affectation	a = b
Addition	a = a + b
Soustraction	a = a - b
Multiplication	a = a * b
Division	a = a / b
Modulo (reste de division)	c = a % b

Opérateurs cryptiques

Addition d'une constante	a += b
Soustraction d'une constante	a -= b
Multiplication	a *= b
Division	a /= b
Incrémenter	a++ et ++a
Décrémenter	a-- et --a

Opérateurs logiques

supérieur à	a > b
supérieur ou égal à	a >= b
inférieur à	a < b
inférieur ou égal à	a <= b
égal à	a == b
différent de	a != b
ET logique	a && b
OU logique	a || b

Opérateurs sur bits

Shift gauche	a = b << c
Shift droite	a = b >> c
ET entre bits	a = b & c
OU entre bits	a = b | c
Complément bit par bit	a = ~ b
OU EXCLUSIF entre bits	a = b ^ c

Opérateurs d'adressage et divers

Adresse	a = &b
Renvoi à un pointeur	a = *b
Conversion d'un type de données	(type_données) variable
Déterminer la taille en octets	sizeof(var)

17.3. Types de données en C

Type de données	Occupation en mémoire (Bits)	Domaine de définition
unsigned char	8	0 à +255
char	8	-128 à +127
unsigned int	16	0 à +65535
int	16	-32768 à +32767
unsigned short	16	0 à +65535
short	16	-32768 à +32767
unsigned long	32	0 à +4.294.967.295
long	32	-2.147.483.648 à +2.147.483.647
float	32	3.4 E-38 à +3.4 E+38
double	64	1.7E-308 à +1.7E+308
near pointer	16	
far pointer	32	

17.4. Structures de contrôle en C

Boucles

Boucle testée en entrée

```
while (condition)
{
      Corps de boucle
}
```

Boucle testée en sortie

```
do
{
      Corps de boucle
} while (condition);
Boucle de comptage
for (initialisation;condition d'exécution de cycle;modification de la
     variable de décompte)
{
      Corps de boucle
}
```

Ramifications

Structure SI

```
if (condition)
{
   Ramification OUI
}
```

Structure SI..ALORS avec possibilité d'imbrication

```
if (condition)
{
   Ramification OUI;
}
else
{
   Ramification NON
}
```

Structure SI..ALORS sans possibilité d'imbrication

```
(condition ? Ramification OUI : Ramification NON)
```

Ramification inconditionnelle

```
goto label
```

Différenciations de cas

```
switch (variable)
{
        case valeur(1) : instructions; break;
        case valeur(2) : instructions; break;
        case valeur(n) : instructions; break;
        default : instructions;
}
```

17.5. Récapitulation des syntaxes

atoi

Fonction : Convertit une chaîne en valeur entière

Format :

```
int atoi(char *string)
```

Prototype :

```
stdlib.h
```

Description : La chaîne sur laquelle pointe string est convertie en une valeur entière. La valeur retournée est la valeur entière.

fclose

Fonction : Ferme un fichier

Format :

```
int fclose(FILE *pointeur_sur_fichier)
```

Prototype :

```
stdio.h
```

Description : Ferme un fichier. La valeur retournée est 0 en cas d'opération réussie, sinon -1 = EOF en cas d'erreur.

fgets

Fonction : Extraction d'une chaîne d'un fichier.

Format :

```
char *fgets(char *string,int n,FILE *pointeur_sur_fichier)
```

Prototype :

```
stdio.h
```

Description : Extrait n-1 caractères du fichier, et les transcrit dans la chaîne "string". L'extraction s'arrête dès qu'est détecté un RETOUR CHARIOT, ou après n-1 caractères. NULL est placé comme dernier caractère de "string".

fopen

Fonction : Ouvrir un fichier, établir une liaison, et définir le mode d'accès.

Format :

```
FILE *fopen(char *nom_fichier,char *mode)
```

Prototype :

```
stdio.h
```

Description : La fonction retourne un pointeur de type FILE. Les valeurs possibles de "mode" sont :

"w" = création et ouverture en écriture
"r" = ouverture en lecture
"r+" = ouverture en lecture et écriture
"a" = Ajouter des données à la fin d'un fichier (append)
"a+" = Ajouter des données.

Si vous ajoutez la lettre "t" au mode, vous définissez le mode texte. "b" met en place le mode binaire.

fprintf

Fonction : Sortie structurée dans un fichier

Format :

```
int fprintf(FILE *pointeur_sur_fichier,char
            *format [,argument,...])
```

Prototype :

```
stdio.h
```

Description : Renvoie le nombre des octets retournés. - voir printf() -

fread

Fonction : Extraire des enregistrements d'un fichier

Format :

```
int fread( void *enregistrement,int nombre_octets,
           int nombre_enregistrements, FILE *pointeur_fichier)
```

Prototype :

```
stdio.h
```

Description : Lit des enregistrements structurés et retourne le nombre d'enregistrements lus. La valeur retournée est 0 dès lors que la fin du fichier a été atteinte ou qu'une erreur s'est produite.

free

Fonction : Libère de l'espace mémoire sur le tas (heap).

Format :

```
void free (void *pointeur)
```

Prototype :

```
alloc.h
```

Description : Libère une plage de mémoire qui avait été réservée à l'aide de *malloc*.

fscanf

Fonction : Extraction structurée d'un fichier.

Format :

```
int fscanf(FILE *pointeur_sur_fichier,char
            *format [,argument,...])
```

Prototype :

```
stdio.h
```

Description : Lit le contenu d'un fichier et retourne le nombre des champs de saisie. - voir scanf() -

fseek

Fonction : Positionne le pointeur associé à un fichier.

Format :

```
int fseek( FILE *pointeur_fichier, long position_relative,
        int à_partir_de_la_position_actuelle)
```

Prototype :

```
stdio.h
```

Description : Le positionnement est toujours réalisé relativement à la position actuelle, en allant vers la fin du fichier, du nombre d'octets spécifié dans position_relative. En cas de positionnement réussi du pointeur, le résultat de la fonction est 0, sinon, il est différent de 0.

ftell

Fonction : Position du pointeur dirigé sur un fichier, exprimée en octets.

Format :

```
long ftell( FILE *pointeur_fichier)
```

Prototype :

```
stdio.h
```

Description : Renvoie une valeur entière de type long. Renvoie -1L en cas d'erreur.

fwrite

Fonction : Ecriture d'enregistrements dans un fichier.

Format :

```
int fwrite( void *enregistrement,int nombre_octets,
            int nombre_enregistrements, FILE *pointeur_fichier)
```

Prototype :

```
stdio.h
```

Description : Ecrit des enregistrements structurés dans un fichier. En cas d'erreur, la valeur retournée est 0.

getc

Fonction : Extraction de caractères d'un flux (stream).

Format :

```
int getc(FILE *pointeur_sur_fichier)
```

Prototype :

```
stdio.h
```

Description : Macro qui renvoie EOF lorsque la fin du fichier est atteinte ou qu'une erreur s'est produite.

gets

Fonction : Extraction d'une chaîne d'un flux.

Format :

```
char *gets(char *string)
```

Prototype :

```
stdio.h
```

Description : La chaîne à extraire sera conclue par RETURN. Dans la chaîne, RETURN est remplacé par NULL. Renvoie la valeur NULL en cas d'erreur, ou lorsque la fin du fichier est atteinte.

getchar

Fonction : Lecture de caractères entrés au clavier.

Format : `int getchar(void)`

Prototype : `stdio.h`

Description : En cas d'erreur, renvoie EOF (valeur -1).

isalpha

Fonction : Classification de caractères. Test de reconnaissance de lettres.

Format : `int isalpha(int caractère)`

Prototype : `ctype.h`

Description : Renvoie 0 pour "faux" et un nombre positif pour "vrai"

isdigit

Fonction : Classification de caractères. Test de reconnaissance de chiffres

Format : `int isdigit(int caractère)`

Prototype : `ctype.h`

Description : Renvoie 0 pour "faux" et un nombre positif pour "vrai"

islower

Fonction : Classification de caractères. Test de reconnaissance de minuscules

Format : `int islower(int caractère)`

Prototype : `ctype.h`

Description : Renvoie 0 pour "faux" et un nombre positif pour "vrai"

isspace

Fonction : Classification de caractères. Test de reconnaissance de Blanc (espace), Tab, Form Feed, Line Feed et Return

Format :
```
int isspace(int caractère)
```

Prototype :
```
ctype.h
```

Description : Renvoie 0 pour "faux" et un nombre positif pour "vrai"

isupper

Fonction : Classification de caractères. Test de reconnaissance de majuscules

Format :
```
int isupper(int caractère)
```

Prototype :
```
ctype.h
```

Description :
```
Renvoie 0 pour "faux" et un nombre positif pour "vrai"
```

itoa

Fonction : Conversion d'une valeur entière en chaîne.

Format :
```
char *itoa(int valeur, char *string, int base)
```

Prototype :
```
stdlib.h
```

Description : Convertit le nombre entier *valeur* en la chaîne *string*. "base" désigne la valeur de base de valeur. Pour "string" il faut réserver une plage suffisante en mémoire. La valeur retournée est le pointeur dirigé sur la chaîne.

malloc

Fonction : Réservation de mémoire sur le tas (heap).

Format :
```
void *malloc(unsigned taille_en_octets)
```

Prototype :
```
alloc.h
```

Description : La valeur retournée est un pointeur dirigé sur la plage de mémoire dans le tas (heap). Renvoie NULL lorsque l'espace mémoire disponible est insuffisant.

printf

Fonction : Sortie structurée à l'écran.

Format :
```
int printf(*char format_string[,argument...)
```

Prototype :
```
stdio.h
```

Description : Renvoie le nombre des octets retournés, ou EOF (c'est-à-dire -1) en cas d'erreur

putc

Fonction : Ecriture de caractères dans un fichier

Format :
```
int putc(int caractère, FILE *pointeur_sur_fichier)
```

Prototype :
```
stdio.h
```

Description : Renvoie EOF (c'est-à-dire -1) en cas d'erreur

putchar

Fonction : Affiche des caractères à l'écran.

Format :
```
int putchar(int caractère)
```

Prototype :
```
stdio.h
```

Description : Semblable à putc

scanf

Fonction : Entrées structurées depuis le clavier dans des champs de saisie.

Format :
```
int scanf(char *format_string[,argument..])
```

Prototype :
```
stdio.h
```

Description : Renvoie, en cas de succès, le nombre des champs chargés.

sprintf

Fonction : Ecriture structurée dans la mémoire de travail

Format :
```
int sprintf(char *string_en_mémoire,char
                    *format_string[,argument..])
```

Prototype :
```
stdio.h
```

Description : Semblable à printf

sscanf

Fonction : Lecture structurée dans la mémoire de travail.

Format :
```
int sscanf(char *string_en_mémoire,char
                    *format_string[,argument..])
```

Prototype :
```
stdio.h
```

Description : Semblable à scanf

strcat

Fonction : Relier des chaînes

Format :
```
char *strcat(char *string_cible,char *string_source)
```

Prototype :
```
string.h
```

Description : Ajoute "string_source" à la suite de "string_cible"

strcmp

Fonction : Comparaison de chaînes

Format :
```
char *strcmp(char *string1,char *string2)
```

Prototype :
```
string.h
```

Description : Renvoie la valeur -1 si *string1* est inférieur à *string2*, 0 lorsque les deux chaînes sont égales, et 1 si *string1* est supérieur à *string2*.

strcpy

Fonction : Copie de chaînes

Format :
```
char *strcpy(char *string_cible,char *string_source)
```

Prototype :
```
string.h
```

Description : Copie "string_source" dans "string_cible".

strlen

Fonction : Déterminer la longueur d'une chaîne.

Format : `unsigned strlen(char *string)`

Prototype : `string.h`

Description : Renvoie le nombre de caractères de *string*, symbole NULL terminal non compris.

tolower

Fonction : Conversion de lettres en minuscules.

Format : `int tolower(int caractère)`

Prototype : `ctype.h`

Description : Macro. En cas d'erreur aucune valeur n'est renvoyée

toupper

Fonction : Conversion de lettres en majuscules.

Format : `int toupper(int caractère)`

Prototype : `ctype.h`

Description : Macro. En cas d'erreur aucune valeur n'est renvoyée

17.6. Commandes d'édition sous Turbo C et C++

Mouvements du curseur

Fonction	Commande
Mot à gauche	<Ctrl>+<A> ou <Ctrl>+<Curseur gauche>
Mot à droite	<Ctrl>+<F> ou <Ctrl>+<Curseur droite>
Début du texte	<Ctrl> + <PgUp>
Fin du texte	<Ctrl> + <PgDn>
Page écran vers le haut	<PgUp>
Page écran vers le bas	<PgDn>
Début de bloc	<Ctrl> + <K> + <B>
Fin de bloc	<Ctrl> + <K> + <K>

Opérations sur des blocs

Marquer le début du bloc	<Ctrl> + <K> + <B>
Marquer la fin de bloc	<Ctrl> + <K> + <K>
Copier un bloc	<Ctrl> + <K> + <C>
Déplacer un bloc	<Ctrl> + <K> + <V>
Effacer un bloc	<Ctrl> + <K> + <Y>
Extraire un bloc	<Ctrl> + <K> + <R>
Ecrire un bloc	<Ctrl> + <K> + <W>
Imprimer un bloc	<Ctrl> + <K> + <P>

Localiser, remplacer, effacer

Effacer une ligne	<Ctrl> + <Y>
Effacer jusqu'à la fin de la ligne	<Ctrl> + <Q> + <Y>
Effacer le mot à droite	<Ctrl> + <T>
Chercher	<Ctrl> + <Q> + <F>
Chercher et Remplacer	<Ctrl> + <Q> + <A>
Répétition de Recherche/Remplacement	<Ctrl> + <L>
Arrêt de Recherche/Remplacement	<Ctrl> + <U>

17.7. Commandes d'édition sous Quick C

Mouvements du curseur

Fonction	Commande
Mot à gauche	<Ctrl>+<A> ou <Ctrl>+<Curseur gauche>
Mot à droite	<Ctrl>+<F> ou <Ctrl>+<Curseur droite>
Début du texte	<Ctrl>+<Origine>
Fin du texte	<Ctrl>+<Fin>
Page écran vers le haut	<PgUp>
Page écran vers le bas	<PgDn>

Opérations sur des blocs

Le marquage est réalisé à l'aide de la souris ou avec la touche Shift combinée avec les touches du curseur.

Fonction	Commande
Effacer un bloc	<Suppr>

Chercher, remplacer, effacer

Fonction	Commande
Effacer une ligne	<Ctrl> + <Y>
Effacer jusqu'à la fin de la ligne	<Ctrl> + <Q> + <Y>
Effacer le mot à droite	<Ctrl> + <T>
Chercher	<Ctrl> + <Q> + <F>
Chercher et Remplacer	<Ctrl> + <Q> + <A>
Répéter Recherche/Remplacement	<Ctrl> + <L>

17.8. Affectation des touches de fonction en Turbo C et C++

Commande	Touches de fonction
Charger un programme	<F3>
Charger l'un des derniers fichiers	<Alt> <F3>
Sauvegarder le fichier courant	<F2>
Lancer un programme	<Ctrl> + <F9>
Relancer un programme au débogage	<Ctrl> + <F2>
Suivi de débogage actif	<F8>
Jusqu'à la position du curseur (Débogage)	<F4>
Mode pas à pas (Débogage)	<F7>
Step over (débogage)	<F8>
Placer un point d'arrêt	<Ctrl> + <F8>
Déterminer/modifier des valeurs de variables (Débogage)	<Ctrl> + <F4>
Ajouter une variable de type Watch	<Ctrl> + <F7>
Ecran utilisateur	<Alt> + <F5>
Quitter le programme	<Alt> + <X>

17.9. Affectation des touches de fonction en Quick C

Commande	Touches de fonction
Charger le dernier fichier	<F2>
Lancer un programme	<F5>
Relancer un programme	<Shift> + <F5>
Suivi de débogage actif	<F8>
Jusqu'à la position du curseur (Débogage)	<F7>
Sauter (débogage)	<F10>
Placer un point d'arrêt	<F9>
Poursuivre la recherche	<F3>
Erreur suivante	<Shift> + <F3>
Erreur précédente	<Shift> + <F4>
Ecran utilisateur	<F4>
Aide contextuelle	<F1>
Vue globale des textes d'aide	<Shift> + <F1>
Quitter le programme	<Alt> + <F4>

17.10. Solutions des exercices

Résolution des exercices du chapitre 2

```
/************************ EX021.C ****************************************/
main()
    {
    printf("D A R I M O N T\n");
    }
```

```
/************************ EX022.C ****************************************/
main()
    {
    printf("D A R I M O N T\n");
    printf("13 mont des planeurs\n");
    printf("\n");
    printf("77777 A I L L E U R S ");
    }
```

```
/************************ EX023.C ****************************************/
main()
{
printf("\nP\t\tJ\nE\t\tE\nT\t\tR\nI\t\tO\nT\t\tM\nJ\t\tE\nE\nA\nN");
}
```

Résolution des exercices du chapitre 3

```
/************************ EX031.C ****************************************/
/* Chapitre 3.10 Exercice 1 */
int i;
main()
{
    for (i=0;i < 10; i = i + 1)
        printf("D A R I M O N T\n" );
}
```

```
/************************ EX0311.C ***************************************/
int i;
main()
{
    for (i=0;i < 10; i = i + 1)
         printf("D A R I M O N T\n" );
}
```

```
/********************** EXO312.C ****************************************/
int i = 0;
main()
{
    do {
        printf("\nD A R I M O N T" );
        i = i + 1;
    } while (i < 10);
}
```

```
/* ******************** EXO313.C ****************************************/
int i = 0;
main()
{
    while (i < 10)
        {
        printf("\nD A R I M O N T" );
        i = i + 1;
        }
}
```

```
/******************* EXO32.C ********************************************/
int i = 0;
main()
{
    for (i=1; i < 11 ; i = i + 1)
        {
          if (i == 3)
          printf("Voici le nombre %d\n",i);
          else
          {
           if (i == 7)
             printf("Voici le nombre %d\n",i);
           else
             printf("Je compte %d\n",i);
          }
        }
}
```

```
/************************ EXO33.C ****************************************/
main()
{
int index;
   for (index = 3;index < 13;index = index + 1)
      switch (index)
      { /* Début SWITCH */
         case 4  :
         case 6  :
         case 8  :
         case 10 :
         case 12 :    printf("La valeur de index est %d\n" , index);
         break;
      }
 }        /*  Fin de main()       */
```

Résolution des exercices du chapitre 4

```
/************************ EX041.C ****************************************/
main()
{
  int x = 11,y = 11,z = 11;
 char a = 40,b = 40,c = 40;
 float r = 12.987,s = 12.987,t = 12.987;
 if (x == y) z = -13;
   printf("La valeur de z est %d\n",z);
 if (x > z)  a = 'A';
   printf("La valeur de a est %d\n",a);
 if (!(x > z)) a = 'B';
   printf("La valeur de a est à présent %c\n",a);
 if (b <= c) r = 0.0;
   printf("La valeur de r est %-2.2f\n",r);
 if (r != s) t = c/2;
   printf("Et à présent est la valeur de t est égale à %-5.2f\n",t);
 if (x = (r != s)) z = 1000;
   printf("La valeur de x est égale à %d et est donc positive,
           c'est pourquoi z est égal à %d\n",x,z);
 if (x = y) z = 222;
   printf("La valeur de x est à présent égale à %d et celle de z est
           égale à %d\n",y,z);
 if (x != 0) z = 333;
   printf("Etant donné que x est différent de 0, z est à présent égal à
          %d\n",z);
 if (x) z = 444;
   printf("Etant donné que x est un nombre positif, à savoir %d, z est
          égal à %d\n",x,z);
 x = y = z = 77;
 if ((x == y) && (x == 77)) z = 33;
   printf("Etant donné que x est égal à y et que x est égal à 77, z est
           égal à %d\n",z);
 if ((x > y) || (z > 12)) z = 22;
   printf("Etant donné que x n'est en effet pas supérieur à y, mais que z
          est supérieur à 12, la valeur de z est à présent %d\n",z);
 if (x && y && z) z = 11;
   printf("Etant donné que x, y et z sont des nombres positifs, z se voit
          attribué la valeur %d \n",z);
 if ((x = 1) && (y = 2) && (z = 3)) r = 12.00;
  printf("x reçoit la valeur %d, y vaut %d, z vaut %d. C'est pourquoi r est
          égal à %2.2f\n", x,y,z,r);
 if ((x == 2) && (y = 3) && (z = 4)) r = 14.56;
  printf("Etant donné que x est égal à 1, toute la condition est fausse, ");
  printf("et ainsi \nr reste égal à %2.2f\n",r);
 if (x == x); z = 27.345;
```

```
  printf("Simple affectation, z est égal à %d\n",z);
 if (x != x)  z = 27.345;
  printf("Etant donné que x ne peut être différent de x, z reste
          inchangé.\n");
 if (x = 0)   z = 27.345;
  printf("Etant donné que la valeur de x est %d et donc \"faux\", ",x);
  printf("z reste inchangé.");
}
```

Résolution des exercices du chapitre 5

```
/************************ EX051.C ****************************************/
int i;
int y = 1;
main()
{
  for (i=1;i<11 ;i++)
  {
     y = i;
     printf("La valeur de i est %d et son carré est %d\n",i,(y*=y));
  }
}
```

```
/************************ EX052.C ****************************************/
int i;
float f,t;
main()
{
    for (i=1;i<11;i++)
    {
        t = i;
        f = 1 /t;
        printf("La valeur de i est %d et son inverse est %f\n",i,f);
    }
}
```

```
/************************ EX053.C ****************************************/
int i;
main()
{
    for (i=1;i<101;i++)
    {
        if ((i>31) && (i < 39))
           printf("La valeur de i est %d\n",i);
    }
}
```

```
/**************************EX054.C ************************************/
main()
{
int valeur;
 for(valeur= 0;valeur < 10;valeur++)
 {
   if (valeur == 2)
     printf("VALEUR est à présent égal à %d\n",valeur);
     (valeur < 5?printf("VALEUR vaut à présent  %d, et est donc
                 inférieur à 5\n",valeur):
  printf("VALEUR vaut à présent  %d, et est donc supérieur à 4\n",valeur));
 }
}
```

```
/****************** EX0551.C ******************************************/
int i;
main()
{
    for (i=0;i < 10; i++)
         printf("D A R I M O N T\n" );
}
```

```
/* ******************** EX0552.C ****************************************/
int i = 0;
main()
{
    do {
            printf("D A R I M O N T\n" );
       } while ((i++) < 10);
}
```

```
/******************** EX0553.C ****************************************/
int i = 0;
main()
{
    while (i++ < 10)
    {
         printf("D A R I M O N T\n" );
    }
}
```

Résolution des exercices du chapitre 6

```
/************************* EX061.C ****************************************/
int fahrenheit(int w)
{
    return(32 + (w * 9) / 5);
}
main()
{
  int index;
  int celsius;
  int f;
  printf("\t\tTable de correspondance Celsius/Fahrenheit\n");
  printf("\t\t\n\n");
  for(index = -2;index <= 12;index++)
  {
     celsius = 10 * index;
     f = fahrenheit(celsius);
     printf("\t\t  C =%4d   F =%4d  ",celsius,f);
     if (celsius == 0)
        printf("\t\tPoint de solidification de l'eau");
     if (celsius == 100)
        printf("\t\tPoint d'évaporation de l'eau");
     printf("\n");
  }
}
```

```
/************************ EX062.C *****************************************/
 void carre(int nombre);
 main()
 {
   int index;
   index = 100;
   carre(index);
 }
 void carre(int nombre)
 {
    printf("Le carré de index = %d est %d\n",
           nombre,(nombre*nombre));
    nombre++;
    if (nombre <111);
        carre(nombre);
 }
```

```
/************************ EX063.C *****************************************/
void compte_a_rebours(int var);
main()
{
   int index;
   index = 8;
   compte_a_rebours(index);
}
void compte_a_rebours(int var)
{
```

```
   var--;
   printf ("La valeur de VAR avant interruption de la récursion est
           \t %d\n",      var);
   if (var > 0)
      compte_a_rebours(var);
   printf("A présent la valeur de VAR est \t %d\n",var);
}
```

Résolution des exercices du chapitre 7

```
/************************* EX071.C ****************************************/
#define CARRE(A) (A) * (A)
#define MAX 5
#define MIN -5
main()
{
    int i;
    i = MIN;
    while ( i <= MAX)
    {
        printf (" Le carré de i = %d est \t %d\n",              i,CARRE(i));
        i++;
    }
}
```

```
/*************************** EX072.C ************************************/
#define FAHRENHEIT(C) ( 32 + (C * 10 * 9) / 5 )
main()
{
int index;
int celsius;
   printf("\t\tTable de correspondance Celsius/Fahrenheit\n");
   printf("\t\t\n\n");
   for(index = -2;index <= 12;index = index + 1)  /* Début de boucle  */
   {
      celsius = 10 * index;
      printf("\t\t  C =%4d   F =%4d  ",celsius,FAHRENHEIT(index));
      if ( (index * 10) == 0)
         printf("\t\tPoint de solidification de l'eau");
      if ( (index * 10) == 100)
         printf("\t\tPoint d'évaporation de l'eau");
      printf("\n");
   }
}
```

Résolution des exercices du chapitre 8

```
/*************************** EX081.C ************************************/
#include <stdio.h>
int longueur_string(char strg[]);
main()
{
  char string[80];
  strcpy(string, "Chaîne_de_caractères");
  printf("\nLa longueur de la chaîne %s est de %d caractères ",string,
         longueur_string(string));
}
int longueur_string(char strg[])
{
    int l = 0 ;
    while (strg[l])
        l++;
    return(l);
}
```

```
/************************ EX082.C ****************************************/
#define BLANC "        "
main()
{
int i = 0,z=0, y=1;
char chemin[80], affichage[80];
    strcpy(chemin,"c:\\dat\\txtdat\\livre");
    strcpy(affichage,BLANC);
    while(chemin[i] != '\0')
    {
       if (chemin[i] != '\\')
       {
          affichage[z] = chemin[i];
          z++;
          i++;
       }
       else
       {
          printf("\nLe nom du répertoire %d est :  %s",y,affichage);
          z=0;
          strcpy(affichage,BLANC);
          y++;
          i++;
       }
    }
    printf("\nLe nom du répertoire %d, qui est le dernier, est :
           %s",y,affichage);
}
```

```
/************************ EX083.C ****************************************/
char mot[80];
int l=0;
main()
{
    printf("Veuillez entrer un mot ");
    scanf ("%s",mot);
    while (mot[++l] != '\0')
          ;
    printf("\nLa longueur du mot entré est de %d caractères ",l);
}
```

Résolution des exercices du chapitre 9

```
/************************ EX091.C ****************************************/
#include <stdio.h>
#define MAX 10
void champ_tri(int tableau_tri[]); /* Prototypes */
void imprime(int tableau_tri[]);
int champint[] = {22,100,1,19,250,8,4,20,30,0}; /* initialisation du
                                                   tableau */
main()
{
    champ_tri(champint);
    imprime(champint);
}
void champ_tri(int tableau_tri[])
{
    int x,          /* boucle extérieure */
        y,          /* boucle imbriquée */
        temp=0,     /* Champ d'aide */
        i;          /* Indice */
    for (x = 0; x < MAX ; x++)
        for(y = 0; y < MAX; y++)
        {
          if(tableau_tri[x] < tableau_tri[y])
          {
             temp = tableau_tri[y];
             tableau_tri[y] = tableau_tri[x];
             tableau_tri[x] = temp;
          }
        }
}
void imprime(int tableau_tri[])
{
    int i=0; /* Indice */
    while(i < MAX) /* Boucle d'affichage */
         {
            printf("\nLe %dième élément dans le champ trié est %d",
                    (i+1),tableau_tri[i]);
         i++;
         }
}
```

```
/************************ EX091.C ****************************************/
#include <stdio.h>
#define MAX 10
void champ_tri(int tableau_tri[],int sens);
void imprime(int tableau_tri[]);
int champint[] = {22,100,1,19,250,8,4,20,30,0};
main()
{
    champ_tri(champint,1);
    imprime(champint);
    champ_tri(champint,2);
    imprime(champint);
}
void champ_tri(int tableau_tri[], int sens)
{
    int x,          /* Boucle extérieure */
        y,          /* Boucle imbriquée */
        temp=0,     /* Champ d'aide */
        i;          /* Indice */
    if (sens == 1)
    {
      for (x = 0; x < MAX ; x++)
        for(y = 0; y < MAX; y++)
        {
            if(tableau_tri[x] < tableau_tri[y])
            {
               temp = tableau_tri[y];
               tableau_tri[y] = tableau_tri[x];
               tableau_tri[x] = temp;
            }
        } /* Fin boucle imbriquée */
      } /* Fin boucle extérieure */
    } /* Fin IF */
    if (sens == 2)
    {
      for (x = 0; x < MAX ; x++)
        for(y = 0; y < MAX; y++)
          {
            if(tableau_tri[x] > tableau_tri[y])
            {
               temp = tableau_tri[y];
               tableau_tri[y] = tableau_tri[x];
               tableau_tri[x] = temp;
             } /* Fin IF */
          } /* Fin boucle imbriquée */
      } /* Fin boucle extérieure */
    } /* Fin IF */
}
```

```
void imprime(int tableau_tri[])
{   int i=0; /* Indice */
    while(i < MAX) /* Boucle d'affichage */
    {
       printf("\nLe %dième élément dans le champ trié est %d",
              (i+1),tableau_tri[i]);
       i++;
    }
}
```

Résolution des exercices du chapitre 10

```
/********************** EXO101.C ****************************************/
#define MAX 80
int output(char *string)
{
    int l = 0;
    while (*string != '\0')
    {
          printf("%c",*string);
          ++l;
          string++;
    }
    return(l);
}
main()
{
    char string[MAX], *p;
    int longueur;              /* déclaration */
    strcpy(string, "Chaîne_de_caractères");
    p = string;
    longueur = output(p);
    printf("\n\nLa longueur de la chaîne est de %d caractères", longueur);
}
```

```
/************************* EXO102.C ***************************************/
#define MAX 80
int copier_string(char *s1, char *s2)
{
    int l = 0;
    while(*s2 != '\0')
    {
         *s1 = *s2;
         s2++;
         s1++;
         l++;
    }
    *s1 = '\0';
    return(l);
}
```

```
main()
{
    char string1[MAX], string2[MAX];
    int l;
    l = copier_string(string1, "1234567890");
    printf("\n%s, a une longueur de %d caractères ",string1,l);
}
```

```
/************************ EXO103.C ****************************************/
#define MAX 80
#define S1 "Pari"
#define S2 "Parisienne"
int test_string(char *s1, char *s2, int precision)
{
    int l = 0;
    while(*s1 != '\0')      /* Tant que la fin de la première chaîne
                               n'est pas atteinte */
    {
    if (*s1 == *s2)   /* Si les caractères sont identiques */
       {
           s1++;              /* Augmenter les adresses */
           s2++;
           l++;               /* Incrémenter le compteur de caractères */
           if (l == precision)
               return(1);     /* l caractères sont identiques */
           continue;          /* suite de la boucle */
       }
       return(0);
    }
      return(0);
}
main()
{
    int l;
    l = 7;
    if (test_string(S1, S2, l))
       printf("\n\n%s et %s sont équivalents sur %d caractères\n\n",
              S1, S2, l);
    else
       printf("%s et %s sont différents sur %d caractères\n\n", S1, S2, l);
}
```

Résolution des exercices du chapitre 11

```
/************************* EXO111.C ****************************************/
#include <stdio.h>
main()
{
    int c;
    while ((c = getch()) != '$')
        printf("\n%c  %d  %x", c, c, c);
}
```

```
/************************ EXO112.C *****************************************/
#include <stdio.h>
void table_ascii(void);
main()
{
    table_ascii();
}
void table_ascii(void)
{
    int i = 31,
        c = 0;
    putchar(13);putchar(10);
    while(i++ < 256)
    {
       printf("%d %c ",i, i);
       c++;
       if ( c == 10)
       {
         putchar(13);putchar(10);
         c = 0;
       }
    }
}
```

```
/********************** EXO113.C ***************************************/
#include <stdio.h>
#define BEEP 7
int get_keyboard(void);
main()
{
    printf("\nLe chiffre entré est %c",get_keyboard());
}
int get_keyboard(void)
{
    int c = 'x';
    while(c)
    {
      c = getch();
      switch(c)
      {
        case '1':
        case '2':
        case '3':
```

```
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9': return(c);
        default : putchar(BEEP);
      }  /* Fin de SWITCH */
    }  /* Fin de WHILE */
} /* Fin de la fonction */
```

```
/********************** EXO1131.C ****************************************/
#include <stdio.h>
#define BEEP 7
int get_keyboard(void);
main()
{
    printf("\nLe chiffre entré est %c",get_keyboard());
}
int get_keyboard(void)
{
    int c = 'x';
    int dummy=1;
    while(dummy)
    {
      c = getch();  /* Mémorisation de touche */
      (c >= '1') && (c <= '9')?dummy = 0:putchar(BEEP);
    }
    return(c);
}
```

Résolution des exercices du chapitre 12

```
/************************ EXO121.C *************************/
#include <stdio.h>
void get_nomfichier(void);
int open_fichier(char *nomfichier, char *statut);
int output(FILE *fp);
char fichier[80];
FILE *pointeurfichier;
main()
{
    get_nomfichier();
    if (open_fichier(fichier, "r"))
       output(pointeurfichier);
}
void get_nomfichier(void)
{
    printf("Veuillez entrer un nom de fichier : -> ");
    scanf("%s",fichier);
}
int  open_fichier(char *nomfichier, char *statut)
{
```

```
    if( (pointeurfichier = fopen(nomfichier, statut))) return(1);
    else
      {
      printf("\n\nLe fichier %s n'a pas pu être ouvert", fichier);
      return(0);
      }
}
int output(FILE *fp)
{
    char c;
    while((c = getc(fp)) != EOF)
    {
      putchar(c);
    }
    return(1);
}
```

```
/************************** EXO122.C ************************************/
#include <stdio.h>
void get_nomfichier(void);
int open_fichier(char *nomfichier, char *statut);
int output(FILE *fp);
char fichier[80];
FILE *pointeurfichier;
main(int argc, char *argv[])
{
    if (argc > 1)
       strcpy(fichier, argv[1]);
    else
       get_nomfichier();
    if (open_fichier(fichier, "r"))
       output(pointeurfichier);
}
void get_nomfichier(void)
{
        printf("Veuillez entrer un nom de fichier : -> ");
        scanf("%s",fichier);
}
int  open_fichier(char *nomfichier, char *statut)
{
    if( (pointeurfichier = fopen(nomfichier, statut)) != NULL)
      return(1);
    else
    {
      printf("\n\nLe fichier %s n'a pas pu être ouvert\n\n", fichier);
      return(0);
    }
}
int output(FILE *fp)
{
```

```
    char c;
    while((c = getc(fp)) != EOF)
    {
      putchar(c);
    }
    return(1);
}
```

Résolution des exercices du chapitre 13

```
/********************** EX0131.C ****************************************/
#include <stdio.h>
#define RETURN 13
#define BEEP 7
int input_enregistrement(void);
void saisie(int l, char *strg);
struct {
       char nom[30];
       char rue[30];
       char ville[30];
       char tel[20];
       } enregistrement;
main()
{
    input_enregistrement();
}
int input_enregistrement(void)
{
    strcpy(enregistrement.nom,"9");
    while(1)
    {
        printf("\nVeuillez entrer votre nom : -> ");
        saisie(30, enregistrement.nom);
        if ( !(strcmp(enregistrement.nom, "x")))
           return(1);
        printf("\nVeuillez entrer votre rue : -> ");
        saisie(30, enregistrement.rue);
        printf("\nVeuillez entrer votre ville : -> ");
        saisie(30, enregistrement.ville);
        printf("\nVeuillez entrer votre numéro de téléphone : -> ");
        saisie(30, enregistrement.tel);
        printf("\n\n%s %s %s %s",enregistrement.nom, enregistrement.rue,
               enregistrement.ville, enregistrement.tel);
     }
}
void saisie(int l, char *strg)
{
    int i = 0;
    int a;
    char c;
        a = l - 1;
        while ( (c = getch()) != RETURN)
        {
```

```
            if (i < a)
            {
                putchar(c);
            *(strg + i) = c;
            i++;
            }
            else
            putchar(BEEP);
        }
    *(strg+i) = NULL;
}
```

```
/************************ EX0132.C ****************************************/
#include <stdio.h>
#define RETURN 13
#define BEEP 7
#define FICHIER "ADR.DAT"
int input_enregistrement(void);
void saisie(int l, char *strg);
int init_fichier(char *nomfichier);
int affichage(FILE *pointeurfichier);
void fin_prog(void);
int get_keyboard(void);
FILE *fp;
struct {
        char nom[30];
        char rue[30];
        char ville[30];
        char tel[20];
        } enregistrement;
main()
{
    int c;
    init_fichier(FICHIER);
    while ( c = get_keyboard())
    {
        switch(c)
        {
         case '1': input_enregistrement();break;
         case '2': affichage(fp);break;
         case '3': fin_prog();
        }
    }
}
void fin_prog(void)
{
    fclose(fp);
    printf("\n\nFin du programme .......");
    exit(1);
}
int init_fichier(char *nomfichier)
{
    if ( (fp = fopen(nomfichier,"a+")) == NULL )
        return(0);
    else
```

```
        return(1);
}
int affichage(FILE *pointeurfichier)
{
    fseek(pointeurfichier,0l,SEEK_SET);
    while(fread(&enregistrement, sizeof(enregistrement),1,pointeurfichier))
    printf("\n\n%s %s %s %s",enregistrement.nom, enregistrement.rue,
            enregistrement.ville, enregistrement.tel);
    getch();
}
int input_enregistrement(void)
{
    strcpy(enregistrement.nom,"9");
    while(1)
    {
        printf("\nVeuillez entrer votre nom : -> ");
        saisie(30, enregistrement.nom);
        if ( !(strcmp(enregistrement.nom, "x")))
           return(1);
        printf("\nVeuillez entrer votre rue : -> ");
        saisie(30, enregistrement.rue);
        printf("\nVeuillez entrer votre ville : -> ");
        saisie(30, enregistrement.ville);
        printf("\nVeuillez entrer votre numéro de téléphone : -> ");
        saisie(30, enregistrement.tel);
        fseek(fp,0l,SEEK_END);
        fwrite(&enregistrement, sizeof(enregistrement),1,fp);
     }
}
void saisie(int l, char *strg)
{
    int i = 0;
    int a;
    char c;
        a = l - 1;
        while ( (c = getch()) != RETURN)
        {
           if (i < a)
           {
              putchar(c);
              *(strg + i) = c;
              i++;
           }
        else
           putchar(BEEP);
        }
        *(strg+i) = NULL;
 }
 int get_keyboard(void)
 {
    int c = 'x';
    printf("\nVeuillez entrer (1), (2) ou (3)  : -> ");
    while(c)
    {
      c = getch();
```

```
      switch(c)
      {
        case '1':
        case '2':
        case '3': return(c);
        default : putchar(BEEP);
      }  /* Fin de SWITCH */
    }  /* Fin de WHILE */
} /* Fin de la fonction */
```

Résolution des exercices du chapitre 14

```
/********************** EXO141.C ****************************************/
#include <stdio.h>
#define FICHIER "LISTE.DAT"
int input(void);
int output(FILE *pointeurfichier);
struct liste
   {
   int nombre;
   struct liste *next;
   } *start, *act, *fin;
FILE *fp;
int size = sizeof(struct liste);
main()
{
    int i;
    input();
    fp = fopen(FICHIER,"w");
    output(fp);
    act = start; /* Libère à nouveau la mémoire */
    fin = act->next;
    while( fin != NULL)
    {
        fin = act->next;
        free(act);
    }
    fclose(fp);
    printf("\nAppuyez sur une touche. Le fichier sera lu puis affiché.");
    getch();
    fp = fopen(FICHIER,"r");
    while( fread(&i,2,1,fp))
        printf("\n%d ",i);
    getch();
    fclose(fp);
}
int input(void)
{
    int i;
    start = (struct liste *) malloc(size);
    fin = (struct liste *) malloc(size);
    printf("\nVeuillez entrer le premier élément  :-> ");
    scanf("%d",&i);
```

```
    start->nombre = i;
    start->next = NULL;
    fin = start;
    fin->next= NULL;
    while(1)
    {
     act = (struct liste *) malloc(size);
     printf("\nVeuillez entrer l'élément suivant de la liste :-> ");
     scanf("%d",&i);
     if ( i == 0 )
        break;
     act->nombre = i;
     fin->next = act;
     act->next = NULL;
     fin = act;
     }
     return(1);
}
int output(FILE *pointeurfichier)
{
    int i;
    act = start;
    while(act != NULL)
    {
      printf("%d ",act->nombre);
      i = act->nombre;
      fwrite(&i,2,1,pointeurfichier);
      act = act->next;
    }
    return(1);
}
```

```
/************************** EXO142 ****************************************/
#include <stdio.h>
#define MAX 12
struct liste
       {
       char identification;
       int age;
       int classe;
       struct liste *next;
       } *start, *act, *fin;
int size = sizeof(struct liste);
main()
{
   int index;
   struct liste garcon, fille;
   start = (struct liste *) malloc(size);
   fin  = (struct liste *) malloc(size);
   start->identification = 'R';
   start->age = 16;
   start->classe = 11;
   start->next = NULL;
   fin = start;
   fin->next = NULL;
```

```
   for (index = 1;index < MAX;index++)
      {
      act = (struct liste *) malloc(size);
      act->identification = 'A' + index;
      act->age = 16;
      act->classe = 11;
      fin->next = act;
      act->next = NULL;
      fin = act;
      }
   act = start;
   while(act->next != NULL)
   {
     printf("%c a %d ans et fréquente la classe %d\n",
        act->identification, act->age,
         act->classe);
     act = act->next;
   }
}
```

Résolution des exercices du chapitre 15

```
/************************* EXO151.C ****************************************/
#include <stdio.h>
void get_nomfichier(void);
int open_fichier(char *nomfichier, char *statut);
int touche_ascii(int c);
char fichier[80];          /* variables globales */
FILE *pointeurfichier;
main()
{
    int c;
    get_nomfichier();
    open_fichier(fichier, "r");
    while ( (c = getc(pointeurfichier)) != EOF)
    {
    if (touche_ascii(c))
         putchar(c);
    }
}
void get_nomfichier(void)
{
    printf("Veuillez entrer un nom de fichier : -> ");
    scanf("%s",fichier);
}
int open_fichier(char *nomfichier, char *statut)
{
    if( (pointeurfichier = fopen(nomfichier, statut)))
      return(1); /* Ouverture de fichier réussie */
    else /* Erreur lors de l'ouverture du fichier */
    {
      printf("\n\nLe fichier %s n'a pas pu être ouvert",
            fichier);
```

```
      return(0);
    }
}
int touche_ascii(int c)
{
    int i;
    return(c > 32?1:0);
}
```

```
/************************** EX0152.C ***************************************/
#include <stdio.h>
#include <ctype.h>
void get_nomfichier(void);
int open_fichier(char *nomfichier, char *statut);
int touche_ascii(int c);
char fichier[80];
FILE *pointeurfichier;
main()
{
    int c; /* Variable monocaractère */
    get_nomfichier();
    open_fichier(fichier, "r");
    while ( (c = getc(pointeurfichier)) != EOF)
    {
    if (touche_ascii(c))
         putchar(c);
    }
}
void get_nomfichier(void)
{
    printf("Veuillez entrer un nom de fichier : -> ");
    scanf("%s",fichier);
}
int open_fichier(char *nomfichier, char *statut)
{
    if( (pointeurfichier = fopen(nomfichier, statut)))
      return(1);
    else
    {
      printf("\n\nLe fichier %s n'a pas pu être ouvert",fichier);
      return(0);
    }
}
int touche_ascii(int c)
{
    if (isalpha(c))
       return(1);
    if ( c == 32 )
       return(1);
    if ( (c==10) || (c ==13) )
       return(1);
    return(0);
}
```

```
/******************** EXO153.C ******************************************/
#include <stdio.h>
void get_nomfichier(void);
int open_fichier(char *nomfichier, char *statut);
int touche_ascii(int c);
char fichier[80];
FILE *pointeurfichier;
main()
{
    int c; /* Variable monocaractère */
    get_nomfichier();
    open_fichier(fichier, "r");
    while ( (c = getc(pointeurfichier)) != EOF)
    {
    if (touche_ascii(c))
          putchar(c);
    }
}
void get_nomfichier(void)
{
    printf("Veuillez entrer un nom de fichier : -> ");
    scanf("%s",fichier);
}
int open_fichier(char *nomfichier, char *statut)
{
    if( (pointeurfichier = fopen(nomfichier, statut)))
      return(1); /* Ouverture de fichier réussie */
    else /* Erreur lors de l'ouverture du fichier */
    {
      printf("\n\nLe fichier %s n'a pas pu être ouvert", fichier);
      return(0);
    }
}
int touche_ascii(int c)
{
    if( (c >31) && (c < 123) )
    {
      switch(c)
      {
       case 64:
       case 91:
       case 92:
       case 93:
       case 94:
       case 95:
       case 96:return(0);
      }
      return(1);
    }
    return(0);
}
```

Index

!

#define 116
#include 117
% 62
%c 124, 135
%d 34
%s 124, 135
& 142, 260
&& 72
'\t' 179
* 56, 143
*/ 35
+ 57
++ 78
– 78
/ 57
/* 35
= 70
== 70
\" 259
\0 . . . . . 123, 259
\b 259
\f 259
\n . . . . . 30 - 31, 259
\r 259
\t 32, 259
^ 260
| 260
|| 72
~ 260

A

Accéder . . . . . .
à un élément de la liste 253
à une variable 154
aux divers champs d'une structure 228
directement à des configurations de bits 260
Adresse d'une variable 141, 144
Affectation 43
de valeurs entières 61
de valeurs à des structures 198
Affichage
de champs de structures 198
des caractères à l'écran 301
du premier caractère d'une chaîne 124
en notation décimale 67
en notation décimale longue 68
en notation hexadécimale 67
en notation octale 67
sans signe relatif 67
sous forme d'adresse mémoire 67
sous forme de chaîne de caractères 67
sous forme de nombre à virgule flottante 67
sous forme de type char 67
AGIS.C. 25
Ajouter
de nouveaux éléments à une liste 240
des éléments en fin de liste 247
des caractères 180
des données complémentaires 191
des valeurs à un pointeur 149
Allocation dynamique de la mémoire 235
Append() 181, 247
Argument 112
Arithmétique des pointeurs 154, 202
ARRAY2.C 128
Arrays 136
ARRET.C. 47
Atoi() 293

B

Backspace 259
Base de données 211
BEEP 172
Bibliothèques de fonctions standard 98
Bit 259
BITOPS.C 260
BITSTRUC.C. 264
BOUCLE.C. 282
BOUCLE.PAS. 281
Boucle 22, 40,291
    testée en entrée à l'aide de for 43
    testée en entrée à l'aide de while 40
    testée en sortie à l'aide de do..while 42
Break . . . . . 47 - 48
Breakpoints 105
Buffer 162

C

CALENDR.C 155
CARACTC.C 257
Caractère de contrôle 31
Caractéristique Watch 105
CARRES.C 88
Carriage Return 164
Case 50
CASE.C. 280
CASE.PAS. 279
Cast 236
CELSIUS.C 55
Chaîne 121, 135, 147
CHAINE.C. 232
Champ 211
    d'initialisation 43
Char 71
CHAROUT.C 180
CHERCHE.C 217
Classification de caractères 298

Clavier 159
Code
de conversion 67
source 20
de contrôle 259
Commentaire 26, 35
COMMENTE.C. 35
Comparaison 46
de chaînes 126, 302
logique 68
Compilateur 17
Compiler 20
CON 159
Condition d'interruption 99
Configuration de bits 259
Console 159
Constante 115
alphanumérique 64
littérale 64
monocaractère 64
symbolique 116, 119
CONTINU.C. 30
Continue . . . . . 47 - 48
Conversion
d'une chaîne en valeur entière 293
d'une valeur entière en chaîne 299
de lettres en majuscules 303
de lettres en minuscules 303
de variables 65
Copie
de chaînes 124, 302
du contenu de structures 199
d'un bloc 304
Croissant() 252
CTYP.C. 77
CTYPE.H 259

D

DATA.C. 276
DATA.PAS. 275
Débogage 104, 108, 110
Débordement de la pile 101
Début de fichier 220
Déclaration 34, 154
    d'une chaîne 122
    des pointeurs 145
Decroissant() 252
Default 51
Défilement d'une liste 252
DEFINE.C 116
DEFTYPE.C 210
Delete() 251
DEPLACE.C. 219
Déplacement
    dans une liste à double liaison 252
    d'un bloc 304
Déroulement ligne par ligne 107
Descripteur de champ 34, 67
    %c 124
    %s 124
    pour le renvoi d'une chaîne 135
Détecter les majuscules et les minuscules 256
DEUXDIM.C 129
DEUXDIM2.C 131
DEUXVOIE.C. 150
Différenciation de cas 49, 293
DOMAINE.C 94
Domaines de définition de variables 93
Données 21
    de type char 64
    de type double 67
    de type float 64
Double renvoi de pointeurs 250
DOWHILE.C 42

E

Echo 162
Ecran 159
Ecriture 191
d'enregistrements dans un fichier 212, 297
d'un bloc 304
dans la mémoire de travail 175
de caractères dans un fichier 181, 300
de chaînes 178
de fichiers structurés 229
structurée dans la mémoire de travail 301
ECRIS.C 178
Editer 20, 27
Edition de programmes C 27
Effacer un bloc 304
Empilement 101
En-tête de programme 24
End Of File 182
ENDING 116
Enregistrement 211
Erreur 28
ET logique 260
Evaluation de comparaisons logiques 73
EX_GOTO.C. 53
Exécution séquentielle 21
Exit() 189
Extraction
d'enregistrements d'un fichier 215, 295
d'un bloc 304
d'une chaîne d'un fichier 294
d'une chaîne d'un flux 297
de caractères d'un flux (stream) 297
de chaînes de caractères 183
structurée d'un fichier 296
Extrémité du fichier 229

F

FACT.C. 102
FACT2.C. 108
Far heap 234
Fclose() 179, 191, 293

Fenêtre
    d'édition 20
    de débogage 109
    Watch 105
Fgets 186, 294
Fichier
    d'en-tête 113, 160
    Fermer 293
    Fin 182
    Include 97, 117
    Ouvrir 179
FICHIER.C 187
Fin
    de chaîne 121
    de liste 238, 253
    de fichier 220
FLOATQR.C 90
FONCT1.C. 84
Fonction 85, 138, 283
    à virgule flottante 90
    de bibliothèque 84
    utilisateur 84
FONCTION.C. 285
FONCTION.PAS. 284
Fopen 179, 294
FOR 43
FOR.C 43
Fprintf() 179
Fread() 215, 229
Free() 241
FRETOUR.C 137
Fscanf() 184
Fseek() 296
Ftell() 221, 296
Fwrite() 297

G

Gestion
de la mémoire 101
des chaînes en C 123
des fichiers 177, 191, 193
dynamique de la mémoire 231, 253
Getc() 183, 297
Getch() 163
Getchar 161, 298
Gets 186, 297
Goto 52

H

Heap 234
HELLO.BAS 270
HELLO3.C. 271

I

IF 45, 71
IFELSE.C. 45
Imbrication de structure 206
IMBRIQUE.C 204
Impression des données 190
IMPRFICH.C. 190
Index 142
Indexer un pointeur 148
Indice 131, 135
Indirection 143
INFINI.C. 272
Initialisation 69
Initialiser des tableaux 128, 151
INMEM.C. 173
Input 102
INPUT.C. 171
Insérer des éléments dans une liste 248

Insert() 248
Installation
de Quick C 2.5 16
de Turbo C 2.0 14
des compilateurs 14
Instruction 26
composée 44
Int 33
INTARRAY.C 127
Interpréteur 17
Interruption de boucle 47
INTRO.C. 20
INVERSE 260
IOSIMPLE.C. 160
Isalpha() 259, 298
Isdigit() 259, 298
Islower() 298
Isspace() 259, 299
Isupper() 299
Itoa() 299

L

Label 52
Lancement de programmes 29
Lecture 191
d'un fichier 181
de caractères entrés au clavier 298
de chaînes de longueur quelconque 170
de lignes 185
de mots 183
de valeurs entières 166
depuis la mémoire de travail 175
d'enregistrement 229
structurée dans la mémoire de travail 301
LECTURE.C 182
Liaison logique 72
Libération
des variables dynamiques 241
de l'espace mémoire sur le tas 295

Library function 84
LIFO 101
Line Feed 164
LIRINT.C 166
LISFICH.C. 215
LISLIGN.C 185
Liste à double chaînage 240
    à simple chaînage 237, 253
    des variables 35
    liée 237, 253
LISTE.C 241
LISTEXTE.C 183
LOGIQUE.C. 68

M

Macro 117, 119
MACRO.C 118
Main 24
MAINC.C 188
MAJMIN.C 255
Malloc() . . . . . 234 - 235
Marquer la fin d'un bloc 304
Masque 35
Mathématique des pointeurs 202
Mémoire
    conventionnelle 234
    réservée à la pile 101
    tampon 162
Mode Step-Over 135
Trace 107
MODIFIE.C. 223
Modifier des enregistrements 223
Module 227
MULTITYP.C 66

N

NOECHO.C. 163
Nombres à virgule flottante 64
NON logique 260
Nouvelle ligne 259
NULL 121, 259

O

Octet 259
Opérande 56
Opérateur 56, 77
    arithmétique 79
    binaire 79
    cryptique 290
    d'adressage 143, 291
    d'affectation 57,
    d'incrémentation 78
    de décrémentation 78
    de négation 70
    de supériorité 70
    ET 72
    logique 73, 260, 290
    mathématique 290, 301
    Modulo 62
    OU 72, 260
    Shift 262
    sur bits 290
    unaire 81

P

Périphérique d'entrée standard 159
Pile 100, 106, 113
POINTER.C 141
POINTER2.C 146
Pointeur 141
    d'enregistrement 219
    de fichier 179
    dirigé sur des valeurs entières 149
    non initialisé 287
POINTINI.C 152
Point d'arrêt 105
Position
    courante du pointeur 229
    du pointeur dirigé sur le fichier 221
Principe LIFO 113

Printf() 25, 31, 300
Priorité d'évaluation 73
PRN 190
Procedure 283
Programme modulaire 223
Prototype 97, 113
Putc() 181, 300
Putchar() 161, 183
Puts() 174

R

RECURSIO.C. 98
Récursivité . . . . . 98 - 99, 113
Registre 96
    DI 96
    SI 96
Réinitialisation du programme 107
RELATION.BAS 271
Relier des chaînes 302
Remettre à zéro toutes les variables du programme 107

Renvoi 22, 45
conditionnel 45
de chaînes partielles 123
de nombres 33
de texte 30
Réservation de mémoire sur le tas 300
Résultat de la fonction 87, 113
Retour chariot 164, 259
RETOUR.C. 133
Return() 113
RETURN.C. 164
RIEN.C. 24

S

Saut
de ligne 31
de page 259
de tabulation 32
Scanf() 153, 166, 301
Séquence escape 31, 259
SHIFT.C. 262
Signal sonore 172
SIMPLE.C. 61
Sizeof 236
Sprintf 173, 301
Sscanf() 174, 301
Stack 100
START 116
STDIO.H 97, 160
Strcat() 126, 136, 302
Strcmp() 126, 136, 302
Strcpy() 125, 136, 302
String 25, 135
STRING1.C 122
STRING2.C. 125
STRINGIN.C. 169
Strlen 303
STRUCDAT.C. 212
Struct 196
STRUCT1.C 196

STRUCT2.C 199
STRUCT3.C 200
Structure 195, 228
définie par le programmeur 205
de contrôle 22
de contrôle liées 55
nommée 205
imbriquées 203
Switch 52
Supprimer des éléments d'une liste 251
Switch 49
SWITCH.C. 49
Symbole NULL de fin de chaîne 172

T

Tableau 136
global 153
de nombres à virgule flottante 128
de structures 199
de valeurs entières 127
pluridimensionnels 129, 136
Tabulation 259
Taille d'un fichier 221
Tas 234
TAXVALAJ.BAS 272
TAXVALAJ.C. 273
TELEPHON.C. 224
Tolower() 303
Toupper() 303
Traitement
linéaire 31
séquentiel 31
Translation de bits 263
Type de données 63, 74, 291
FILE 179
Union 207
Short int 67
Typedef 210
TYPESUPP.C. 63

U

Union 208
UNION1.C 207
UNNOMBRE.C. 33

V

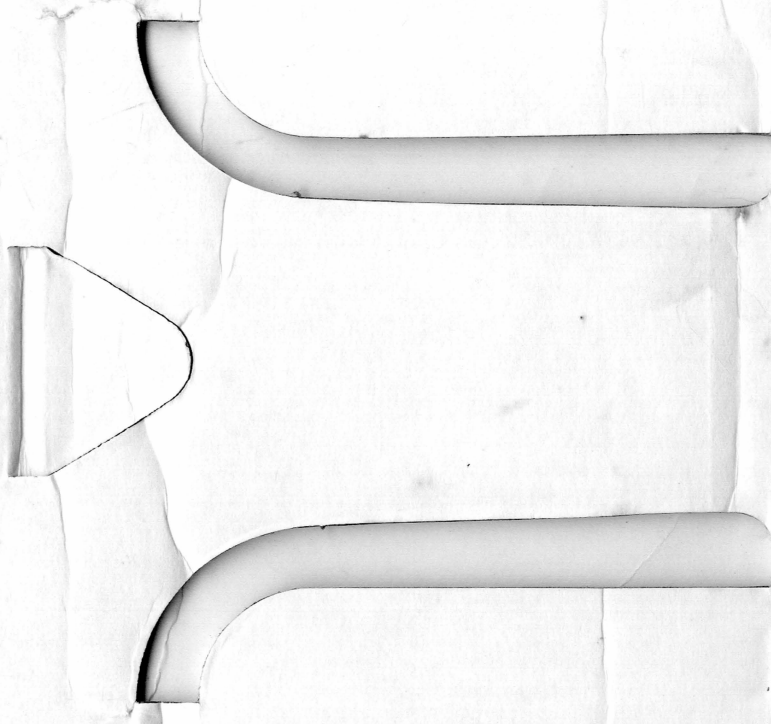
Valeur de vérité 71
Variable
    automatique 95
    d'incrémentation 41
    de boucle 41
    de type entier 33
    de type long int 67
    de type registre 96
    dynamique 231, 253
    externe 95
    fausse 72
    globale 91, 95, 113
    locale 96, 113
    vraie 72
Visualiser le contenu de la pile 107
Void 113

W

WHILE 40
WHILE.C. 40

Achevé d'imprimer
sur les presses de l'imprimerie Maulde et Renou
Tél.: 49.26.14.00
Dépôt légal : Octobre 1993
245/49547

NE PEUT ETRE UTILISEE SANS LE LIVRE



MICRO APPLICATION
PROGRAMMER
EN LANGAGE C
Consulter le Fichier LISEZMOI.DOC
0845
PC ET COMPATIBLES
(C)MICRO APPLICATION
1992
720Ko

UTILITAIRES
SYSTEME D'EXPLOITATION
PROGRAMMATION
TRAITEMENT DE TEXTE/PAO
BASE DE DONNEES
TABLEURS
INTEGRES
CAO / DAO
MATERIEL
LOISIRS ET JEUX
AUTRES

PROGRAMMER EN LANGAGE C

Albrecht DARIMONT

Débutants et programmeurs, voici l'outil idéal pour acquérir de solides connaissances en langage C. Découvrez les structures de contrôle, les opérateurs, les pointeurs et la déclaration des données complexes. Si vous possédez Turbo C, C++ ou Quick C, apprenez à maîtriser les techniques de programmation avancées que sont le traitement des listes, la gestion dynamique de la mémoire ou le contrôle des entrées/sorties. Bénéficiez d'exercices corrigés à chaque chapitre et disposez également en annexe de listes complètes des mots-clés en C, des opérateurs, des instructions et de leur syntaxe, des commandes d'édition, des touches de fonction des compilateurs Turbo C, C++ et Quick C...

Parmi les sujets étudiés:

- Notions de base.
- Structures de contrôle.
- Affectation de valeurs et comparaisons logiques.
- Opérateurs spéciaux.
- Fonctions.
- Constantes et macro-commandes.
- Chaînes et tableaux.
- Liens entre données et fonctions.
- Pointeurs.
- Gestion des entrées/sorties et des fichiers.
- Types de données complexes.
- Gestion dynamique de la mémoire et traitement des listes.
- Manipulation des caractères et des bits.
- Différences entre le C, le Pascal et le Basic.

Réf. 0845. Prix: 195 FF avec la disquette 3"1/2.
ISBN : 2-86899-845-3/ISSN : 0980-1928

MICRO APPLICATION
58 RUE DU FAUBOURG POISSONNIERE
75010 PARIS TEL. (1) 47 70 32 44

09301